RGML: A Markup Language for Characterizing Requirements Generation Processes

Ahmed Samy Sidky, James D. Arthur
Department of Computer Science, Virginia Tech
{asidky, arthur}@vt.edu

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.

1. Introduction

Only over the past ten years has the real importance of requirements engineering been recognized. Subsequently, researchers and practitioners have begun to identify and follow a more systematic and structured approach towards the generation of requirements. Pressman [9] notes that requirements engineering has now been partitioned into five distinct phases: Requirements Elicitation, Requirements Analysis, Requirements Specification, Requirements Validation, and Requirements Management. Similar to the stepwise refinement that has occurred in the past for those software development phases that succeed requirements engineering in the lifecycle development process, we are now seeing the emergence of methodologies, tools, and techniques to support the production of quality requirements. Several requirements generation process models that support the production of quality requirements include the RGM [2], the Agile Requirements Model [1], the Requirements Triage [4], the Knowledge Level Process Model [7] and Volere [10].

However, despite a decade of research within the field of requirements engineering, and the commensurate advancements that come with it, we are still generating requirements that vary widely in quality [13]. What is even more disquieting is that a variance in quality is being observed even when the same requirements generation process is being used. Two prominent sources contributing to the above problem are:

- The requirements engineer either purposefully or inadvertently deviates from the prescribed requirements generation process.
- We are using requirements generation processes that are often composed of an ad hoc collection of sub-processes, activities and tools that are often incompatible with each other.

Clearly, there are several approaches to addressing the sources of these problems. The one we envision, however, is to provide the requirements engineer with an *interactive environment* that guides him/her through an integrated set of structured activities that reflect a unified, well-defined requirements generation process. The product of which will be requirements that consistently reflect quality characteristics.

However, as a first step towards the development of such an environment, we need a mechanism by which we can formally specify the detailed characteristics of the requirements generation process that underlies it. That formal specification can then be used a blueprint to build a corresponding environment. Our ultimate goal, however, is to provide that interaction through direct interpretation of those formal specifications. Clearly, the specification mechanism must be flexible enough to capture the variations among requirements generation models, e.g., 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. In this paper we present the Requirements Generation Markup Language (RGML). The RGML is flexible, expressively powerful and supports the detailed characterization of those elements necessary to and found within requirements generation processes.

The organization of the paper is as follows. In Section 2 we briefly describe several language categories that are “cousins” to RGML. An overview of RGML and its highest-level components are presented in Section 3. Sections 4 and 5 elaborate on those high-level structures. More specifically, Section 4 presents the four control flow structures – sequence, iteration, parallel and ad hoc. Section 5 provides a detailed description of how activities are integrated within the process flow and describes components that comprise activities, e.g., application instantiation and artifact production/consumption. Section 6 describes how, within RGML, we use milestones to impose temporal and causality constraints, and to effect activity alternatives. Finally, in Section 7 we provide an outline of future work that places the RGML in the context of an interactive requirements engineering environment.

2. Background

To date we have no evidence indicating the existence of a specification language for fully characterizing a requirements generation process. What do exist however, are related or “cousin” languages that support similar functionality and objectives in specific domains. We introduce a few of those languages in the remainder of this section.

In the field of requirements engineering, there are many languages that focus on the specification of requirements. Albert II [5] is a formal specification language based upon real-time temporal logic. Albert II is used only to formalize and specify the actual requirements produced from the requirements generation process. Similarly, there is the object-oriented Two-Level Grammar (TLG) [11] that supports the specification of requirements using a natural language style. Requirements Specification Language (RSL) [6], another requirements specification language, is more generic in that it supports the characterization (from a requirements perspective) of the more generic software products and systems. Each of these languages, and many other similar ones, focus on the actual requirements, e.g., how to specify them, how to describe their hierarchies, etcetera.

There are of course languages that focus on the description of processes. The Process Specification Language (PSL) [3] and the XML Process Definition Language [8] are among them. We consider these to be related to RGML because both employ the XML-like structure for specifying process characteristics. They differ from the RGML in that they are intended to support the specification of domain-independent processes – the RGML focuses its expressive capabilities exclusively on processes supporting the generation of requirements. This focus enables RGML to more succinctly describe the process of requirements generation.

Finally, because the syntax of RGML is modeled after markup languages, we will outline similarities and differences between it and other markup languages. In general, the similarities lie in the syntactic format each employs and in their interpretive qualities. What distinguishes them, however, is their domain focus. For example, one of the most common markup languages is the HyperText Markup Language (HTML). One of HTML’s primary objectives is to provide a textual description of graphical entities that can be interpreted and displayed over the Internet. The WML (Wireless Markup Language) is also interpreted to produce a graphical user interface for cell phones. MathML (Mathematical Markup Language) supports the textual description of mathematical notations and the graphical display of their corresponding representations.

Similar to the above markup languages, RGML will be interpreted to provide a graphical representation of and iconic relationship of components within a requirements generation processes. However, differing from the other markup languages the RGML also supports

- the stipulation of protocols and guidelines that lead the requirements engineer through the requirements generation process,
- the of the production and consumption of artifacts throughout the generation process,
- the automatic instantiation of software applications,
- constrained execution of activities through the specification of pre-conditions and milestones,
- the association of templates with artifacts, and
- the specification of conditionals to guide alternative activity paths.

3. RGML Overview

We have described the RGML as a markup language that can be used to formally describe a requirements generation process by characterizing inherent structures, artifacts, and activities. Of equal significance is that it also supports the characterization of a requirements generation process that embodies

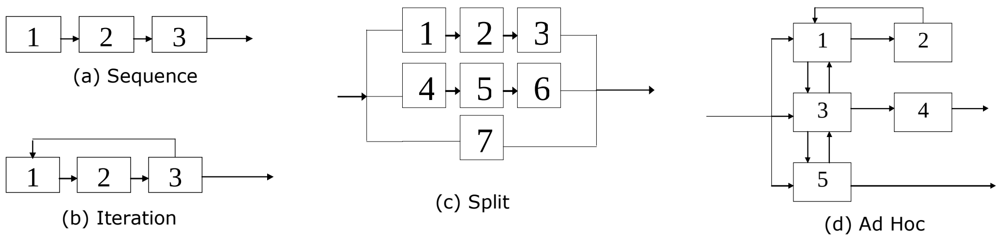

Figure 1: Basic Constructs of Process Structure

- selective control flows and activities based on temporal and conditional specifications, and
- the automatic instantiation of applications for producing artifacts from designated templates.

Any requirements generation process hosts many sub-processes (or activities). The experiences and advice provided by Young [12] were instrumental in the design of the RGML, and in particular, in balancing the process characterization capabilities while maintaining the requisite expressive power and flexibility. According to Young, "A process is a set of *activities* that result in the accomplishment of a task and the achievement of an outcome." Young also states "A process integrates people and tools as well as procedures and methods." Guided by these statements we can divide any process into two major components:

- a group of activities in which each activity is supported by specific tools and procedures, and
- the structure (or the *flow*) of those activities that, when followed, will lead the process to a desirable outcome.

Because the objective of the RGML is to model and characterize processes, we constructed it using similar divisions. As illustrated in Example 1, The RGML consists of two major components, and a minor one. The first major component is structure-oriented and supports the description of the flow of the control in the process. Sequence, iteration and branching are the basic control flow structures. The second major component is activity oriented and provides for the description of individual activities within the requirements generation process. That description includes a characterization of the activity itself, an indication as to which templates it might use, which applications are to be instantiated (and when), alternatives activities, and what preconditions must be satisfied before executing an activity.

The minor component of the RGML is the "Definitions" section. This component is like a dictionary or a repository for template and artifact characteristics. The RGML, like any other markup language, is based on the concept of tags and attributes. As illustrated below, tags divide the language specification into three sections representing the three principal components of the RGML.

```
<RGML>
      <Structure>
            The Flow of control of the activities are nested here
      </Structure>
      <Details>
            The description of the activities are nested here
      </Details>
      <Definitions>
            All the templates, artifacts and types are nested here
      </Definitions>
</RGML>
```

Example 1: Fundamental RGML Structures

As illustrated in the above code, the language has three main elements or tags (<Structure>, <Details>, <Definitions>) nested within the main <RGML> tag. All other tags in the language are nested within one of these main tags.

The next two sections provide a more detailed description of the three language divisions. In particular, Section 4 describes the <Structure> component and how its enclosed tags support flow of control characterization. Section 5 then presents how to use the <Details> and <Definitions> components to construct activities, define and instantiate applications, and integrate artifact and template definitions.

4. Using RGML for Flow of Control

Within a requirements generation process, the flow of control is captured by identifying and characterizing the different control flow structures within that process. That characterization is expressed in the <Structure> section of the RGML language. More specifically, the characterization (or description) provided in the <Structure> section is not intended to define the details of individual process activities. Instead, it is only responsible for the description of the arrangement in which those activities are organized and the sequence of their execution.

The expressive power of the RGML enables one to describe many different layouts or organizations for the activities of a requirements generation process. To achieve this, the RGML uses four orthogonal constructs: Sequence, Iteration, Split and Ad Hoc (see Figure 1). When used together, either in a sequential or in an embedded fashion, they provide the flexibility to describe a large variety of activity sequences.

4.1 Control Flow Constructs

Before describing the 4 basic control flow constructs, we point out that the activities embedded within each construct is denoted by an activity tag and unique name. The details of that activity are defined in the <Details> section.

Figure 1 provides an illustration of the basic structure constructs supported by the RGML. The *sequence construct* (Figure 1-a) is used to express a group of activities executed one after the other. The code shown in Example 2 illustrates the structure depicted by Figure 1-a.

```
<Structure id=main>
          <Sequence>
                    <Activity id=1> Number 1 </Activity>
                    <Activity id=2> Number 2 </Activity>
                    <Activity id=3> Number 3 </Activity>
          </Sequence>
<Structure>
```

Example 2: The Sequence Construct

The *Iteration* construct (Figure 1-b) is used to represent a group of activities executed multiple times until the specified exit criteria are met.

The exit criteria for the loop are specified within the <Exit> tag range. For expressive power and flexibility the RGML supports a variety of ways to express exit criteria. One method is to prompt the user with a "Yes/No" question. Depending on the response, the iteration is continued or terminated. To some extent we view this approach as being "manual" because it requires a response from the user. The specification of automated methods is also supported by the RGML. Fore example, exit criteria could be specified in terms whether or not a specific artifact has been produced, or if a specific milestone (described later) has been triggered. The RGML also permits one to combine different criteria using the Boolean operator tags: <AND> and <OR>.

An RGML example illustrating an iterative specification is provided in Example 3. One notable part of the example is how it uses a combination of three different Boolean conditions to characterize the exit criteria. That is: *Exit the loop if Artifact "A008" has been produced* ***or*** *(if milestone "M1" has been reached* ***and*** *the user answers yes to the question: "Have you met with the board of directors yet?").* Clearly, the inclusion of the logical operators, **and** and **or**, in the definition of RGML enhances its expressive capabilities.

```
<Structure id=main>
   <Iteration>
      <Activity id=1> Number 1 </Activity>
      <Activity id=2> Number 2 </Activity>
      <Activity id=3> Number 3 </Activity>
         <Exit>
            <or>
              <Criteria type="Artifact">A008</Criteria>
                 <and>
                  <Criteria type="Milestone">M1</Criteria>
                  <Criteria type="Question" Exit="Yes">Have you
                  met with the board of directors yet ?</Criteria>
                 </and>
            </or>
         </Exit>
    </Iteration>
<Structure>
```

Example 3: The Iteration Construct

The *split construct* (Figure 1-c) is used to represent a group of activities that can be executed simultaneously or conditionally. The Split element has an important attribute named "Parallel". If "Parallel" is set to "Yes", this enables the flow of execution to flow across all or any subset of the enclosed routes *simultaneously*. However, if the attribute "Parallel" is set to "No" then the flow can only go through one branch, if effect, imposing a selection capability. Independent of whether "Parallel" attribute is set to "Yes" or "No", the sequence (or set of sequences) can be selected manually, or automatically selected using predefined conditionals.

Note that in Figure 1-c the top and middle branches of the structure have multiple activities in sequence, and the last branch has only one activity. As shown in Example 4, we express this configuration in the RGML by nesting two <Sequence> tags within a <Split> tag. Each of the <Sequence> tags will nest their own activities inside. The solo activity found on the last branch will be nested directly under the <Split> tag.

```
<Structure id=main>
    <Split Parallel="Yes">
        <Sequence>
           <Activity id=1> Number 1 </Activity>
           <Activity id=2> Number 2 </Activity>
           <Activity id=3> Number 3 </Activity>
        </Sequence>
        <Sequence>
           <Activity id=4> Number 4 </Activity>
           <Activity id=5> Number 5 </Activity>
           <Activity id=6> Number 6 </Activity>
        </Sequence>
        <Activity id=7> Number 7 </Activity>
     </ Split >
<Structure>
```

Example 4: The Split Construct

The last of our structure constructs is the *"Adhoc"*. This construct is needed to provide the capability to characterize process structures that simply do not conform to sets of sequences, iterations and splits. To achieve that necessary flexibility we also had to be able to view/define control flows as a combination of entry points, exit points and connections among sets of activities.

- Entry points are activities through which we can initiate the flow of control *into* an ad hoc structure.
- Exit points are activities through which the flow of control *exits* an ad hoc structure.
- Connections are used to define flow of control between any two activities *within* an ad hoc structure.

Defining a process structure like that depicted in Figure 1-d would be difficult (if not impossible) using only our basic set of 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) substantially complicate, or negate, the use of a basic construct. Using an "Ad Hoc" construction approach, we can express the combination of activities and control flows as a single structure. In the RGML we expresses the structure found in Figure 1-d as follows:

```
<Structure id=main>
    <Adhoc>
       <Entry>
            <Activity id=1> Number 1 </Activity>
            <Activity id=2> Number 3 </Activity>
            <Activity id=3> Number 5 </Activity>
      </Entry>
      <Connections>
         <Iteration>
            <Activity id=1> Number 1 </Activity>
            <Activity id=2> Number 2 </Activity>
         </Iteration>
         <Iteration>
            <Activity id=1> Number 1 </Activity>
            <Activity id=3> Number 3 </Activity>
         </Iteration>
         <Iteration>
            <Activity id=3> Number 3 </Activity>
            <Activity id=5> Number 5 </Activity>
         </Iteration>
         <Sequence>
            <Activity id=3> Number 3 </Activity>
            <Activity id=4> Number 4 </Activity>
         </Sequence>
       </Connections>
       <Exit>
            <Activity id=4> Number 4 </Activity>
            <Activity id=5> Number 5 </Activity>
       </Exit>
     </Adhoc>
</Structure>
```

Example 5: The "Ad Hoc" Construct

As illustrated in the code of Example 5, the code within the <Adhoc> tag is divided under three main tags.

- The <Entry> tag: All activities that are entry points into the ad hoc structure are identified within the delimiters of the <Entry> tag.
- The <Connections> tag: This tag defines control flow between any two activities.
- The <Exit> tag: All activities that are exit points out of the ad hoc structure are identified within the delimiters of the <Exit> tag.

In the following section we employ the basic constructs defined in this section to characterize Davis' process of Requirements Triage [4].

4.2 An Example: Expressing Requirements Triage in RGML

Figure 2 shows the structure of Davis' Requirements Triage. Code Example 6 (provided below) illustrates how RGML can be used to characterize this structure.

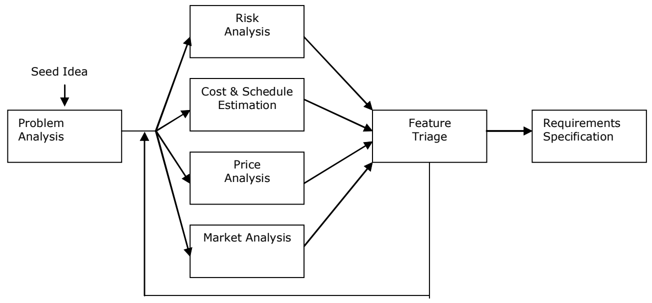

Figure 2: Requirements Triage

```
<Structure id=main>
   <Sequence>
     <Activity id=PA>Problem Analysis</Activity>
     <Iteration>
        <Split Parallel=yes>
            <Activity id=RA>Risk Analysis</Activity>
            <Activity id=CS>Cost and schedule</Activity>
            <Activity id=PAN>Price Analysis</Activity>
            <Activity id=MA>Market Analysis</Activity>
        </Split>
        <Activity id=5>Feature Triage</Activity>
        <Exit>
            <Criteria type="Question">Are you sure…. ?</Criteria>
        </Exit>
    </Iteration>
    <Activity id=RS>Requirements Specification</Activity>
    </Sequence>
</Structure>
```

Example 6: Codifying Requirements Triage

We start with the main <Structure> tag. The first activity, problem analysis, is nested under a <Sequence> tag, followed by an <Iteration> tag and another activity at the end of the specification. It would be invalid to put the <Activity> tag directly under the <Structure> tag because it has to be nested within one of the basic constructs used for structures. In this example we also nest two structures within each other (the split and the iteration). Before closing the <Iteration> tag we have to state the exit criteria for the iteration. In our example we define a question the user must answer, and use that answer to determine whether or not the exit criteria has been satisfied. Because we are focusing on the flow of control and activity identification, we are only showing the structure component of the RGML – we have not yet addressed the specification of the details of each activity. In the next section we will describe how we those activity details are defined.

5. Using RGML to describe Activities

During the description of the process structure, whenever there is a need to refer an activity we referenced it using the <Activity> tag and the activity's unique "id" attribute. In this section we describe the second major component of the RGML, the <Details> section. In particular, we outline how the RGML is used to characterize: the steps comprising a requirements generation activity, how applications are instantiated, and how we specify the production/consumption of the artifacts.

5.1 Defining the Details of an Activity

The activities' descriptions are nested under the <Details> tag. To better illustrate how the RGML addresses the many aspects of an activity, we will present a fabricated example, Example 6, starting with the <Details> tag. This example will also be used to describe other important concepts present later in this paper. Since this is just a contrived example, the details for some tags are omitted.

As shown on line 2 of Example 6 the description of a specific activity is nested within an <Activity> tag having a unique ID of 1. (The "StartUseIf" attribute used in the <Activity> tag supports temporal control and will be discussed in Section 6.) The name of the activity is nested between the <Name> tags, and similarly the goal of the activity is nested between the <Goal> tags (lines 3 and 4). The goal of an activity is a high-level, text-based description of the enclosing activity – no special formulas or methods are required to express the goal. (Pre-conditions, shown in lines 5-8, can be specified during the description of the activity, but we will again defer discussion of this concept until Section 6.)

All activities have primary execution scenarios; they can also have alternative ones. By default, the primary scenario is usually executed (line 9-24 in Example 6). Pre-conditions and milestones, described in Section 6, determine if the primary activity is executer, or which, if any of the alternate activity scenario are executed. Lines 25-45 depict examples of alternative scenarios.

The primary (and alternative) execution scenarios for an activity are expressed using the <Flow> tag. Inside the <Flow> tag we describe the steps that constitute an activity.

```
1.    <Details>
2.     <Activity id=1  StartUseIf=M3>
3.       <Name> Activity Name </Name>
4.       <Goal> This activity is aimed to do so and so </Goal>
5.       <Preconditions>
6.          <Notes> Make sure that so and so happened </Notes>
7.          <Artifact id="MMM" required=true>
8.       </Preconditions>
9.       <Flow>
10.        <Step Required=true>
11.           <Name>123</Name>
12.           <Description> Description in English </Description>
13.        </Step>
14.        <Milestone id=M1>
15.        <Step>
16.           <Name></Name>
17.           <Description> Description in English </Description>
18.           <Action mode=C >
19.              <Type>E-mail</Type>
20.              <Artifact id="ABCD">
21.              <Description></Description>
22.           </Action>
23.        </Step>
24.     </Flow>
25.     <Alternatives>
26.        <Alternative id=3a>
27.          <StartUseIf>M2</ StartUseIf >
28.          <Flow>
29.             <Step required=true>
30.                <Name>654</Name>
31.                <Description> English Description </Description>
32.             </Step>
33.             <Milestone id=M2>
34.             <Step>
35.                <Name></Name>
36.                <Description></Description>
37.                <Action mode=D >
38.                   <Type>E-mail</Type>
39.                   <Artifact id="ABCD">
40.                   <Description></Description>
41.                </Action>
42.             </Step>
43.          </Flow>
44.        </Alternative>
45.     </Alternatives>
46.   </Activity>
47.  </Details>
48.  <Definitions>
49.      <Templates>
50.          <Template id="Meeting">c:\meeting.doc</Template>
51.      </Templates>
52.      <Types>
53.          <Type id="E-mail" app="C:\outlook.exe" />
54.          <Type id="Document" app="c:\winword.exe"/>
55.          <Type id="Spreadsheet" app="c:\excel.exe" />
56.          <Type id="Presentation" app="c:\powerpoint.exe" />
57.          <Type id="Context Diagrams" app=""/>
58.          <Type id="SRS Checker" app="c:\checker.exe">
59.      </Types>
60.      <Artifacts>
61.          <Artifact id="ABCD">
62.              <Name></Name>
63.              <Template>Meeting Request</Template>
64.              <Notes></Notes>
65.          </Artifact>
66.      <Artifacts>
67.  </Definitions>
```

Example 6: RGML <Details> Code

The only tags that can be nested directly within the <Flow> tag are the <Step> tag and the <Milestone> tag. The steps within an activity are expressed by using the <Step> tag, and are nested within the <Flow> tag (lines 10,13 and 15).

The RGML can indicate the importance of different steps by using the "Required" attribute with the <Step> tag. When the "Required" attribute is set to "True", this means that this step is absolutely necessary during the execution of the activity. On the other hand, if the "Required" attribute is set to "False", the requirements engineer can elect to either execute the step, or skip it, depending on current objectives and situational status.

Note that each step in an activity can also have embedded <Name> and <Description> tags. They support the inclusion of descriptive text. To promote specification flexibility, the <Step> tag can also embed additional <Action> tag (and subsequently additional <Step> tags).

5.2 Instantiation of applications and the production of artifacts Using RGML.

In the previous section we discussed how the <Step> is used in the description of an activity. The RGML has more complex features that can also be used in describing activities. We discuss two of those features in this section: the instantiation of applications, and the production of template-driven artifacts.

An <Action> tag that is nested within a <Step> tag is used to indicate that an action is to occur in that specific place during the execution of the activity. Most often, that action engenders the instantiation of an application, e.g., an E-mail program, a Word Processor, or Spreadsheet application. The <Type> tag on lines 19 and 38 designate such instantiations.

Instantiated applications often produce artifacts. If an artifact is to be produced then an <Artifact> tag is embedded within the <Action> tag. All artifacts are defined using the <Definitions> tag. As illustrated in Example 6 artifact "ABCD" referenced on line 39 is defined (between lines 61 and 65) within the <Definitions> tag.

Artifact production, however, prompts the following questions. How will the process handle an artifact if the action is executed several times – for example, in an iteration situation? Should the newly instantiated process (a) create a different version of the artifact each time the activity is executed, (b) should it create the artifact only once, or (c) should it modify the existing artifact. To address this issue we have associated a "mode" attribute with the <Action> tag. The "mode" attribute specifies how to handle the artifact produced in the case of multiple executions of that activity. The mode can take any of the following values:

- *CD (Create and Display)*: Indicates that the process should create the artifact during its first instantiation. If additional instantiations occur, the artifact is to be retrieved and displayed to the user in read-only mode.
- *CM (Create and Modify)*: Indicates that the process should create the artifact during its first instantiation, and if additional instantiations occur, the artifact is to be retrieved, displayed to the user, and be modifiable.
- *C (Create Only)*: Indicates that each time this action is invoked, a new instance of the artifact is to be created. To maintain unique artifact names, the iteration number is appended to the name (id) of the artifact.
- *D (Display Only)*: Indicates that this artifact already exists and is to be displayed in read-only mode.
- *M (Modify)*: Indicates that this artifact already exists. It is displayed to user and the user can modify it.

One additional capability provided within the RGML is associating templates with artifacts. For example the artifact, "ABCD" used in our example is associated with a "Meeting Request" Template (see line 63 of Example 6). Similar to the definition of software applications, templates are also defined within the <Definitions> tag. The "meeting request" template referenced in line 63 is defined between lines 49 and 51.

6. The Conditional Execution of Activities

Within a requirements generation process, the execution of activities is often predicated on the presence of one or more conditions. In this section we discuss three methods provided by the RGML that permit the specification of the conditional execution of an activity.

6.1 Pre-Conditions

Pre-conditions can be specified for any activity. Pre-conditions stipulate what conditions must be present for the enclosing activity to execute. In reality, we have chosen to implement "hard" and "soft" pre-conditions.

One example of using pre-conditions can be seen embedded between lines 5 and 8 in Example 6. The definition of a precondition begins with the <Precondition> tag. In this example the condition is specified using the <Artifact> tag and an associated "Required" attribute. The use of this tag signifies that a specified artifact *must* exist before we can proceed with this activity. If the existence of the artifact *is* required, then the "Required" attribute is set to *true*. If on the other hand, "Required" is set to *false*, then (a) the activity stops, (b) an appropriate message is displayed to the user, and (c) the user must either create the artifact or actively override the condition before the activity execution can proceed.

6.2 Milestones

The RGML also uses the concept of milestones to help express the temporal aspect of a requirements generation process. Milestones are descriptors used to describe an instance separating two states. Each milestone has a scope (or visibility) defined within the <MilestoneScope> tag. A Milestone is like a Boolean variable in a program – it can either be "on" or "off". A milestone can be triggered "on" or "off" at any point within its scope. Triggering a milestone usually indicates the occurrence of a certain event. When the point of an activity's execution moves out of the scope of a milestone, it is automatically reset to an "off" state. For example, in the case of iteration, each time the execution leaves the milestone's scope and re-enters, the milestone's value is reset to "off".

Milestones provide significant flexibility and expressive capabilities. Milestone uses include

- Controlling the execution sequence between non-sequential items,
- Serving as exit criteria for iterations, and
- Controlling when and which alternative version of an activity is to be used.

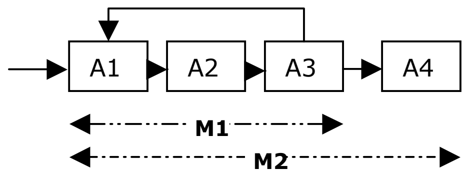

Figure 3: Structure with Milestone Scopes

In the RGML the definition of milestones are given within the <Structure> tag. Milestones are defined using the <MilestoneScope> tag. Each <MilestoneScope> tag must have a unique "Id" attribute. The scope of a milestone is anything in between the opening <MilestoneScope> tag and its closing counterpart. Figure 3 depicts a structure in which two milestones are defined. Milestone M1 has a scope of the iteration structure only; Milestone M2 has a scope of the complete structure shown. Code Example 7 provides a specification of what is depicted in Figure 3.

```
<Structure id=main>
   <Sequence>
     <MilestoneScope id=M2>
        <Iteration>
           <MilestoneScope id=M1>
              <Activity id=A1> A1</Activity>
              <Activity id=A2> A2</Activity>
              <Activity id=A3> A3</Activity>
           </MilestoneScope id=M1>
              <Exit>
                 <criteria type=Question>Have you …. ?</criteria>
               </Exit>
        </Iteration>
        <Activity id=A4> A4</Activity>
     </MilestoneScope id=M2>
  </Sequence>
</Structure>
```

Example 7: Milestone Specification

After defining a milestone within the <Structure> tag, Milestone triggers can then be placed anywhere within the steps of an activity that has been defined within the scope of that milestone.

```
1.   <Details>
2.     <Activity id=A1>
3.       <Flow>
4.         <Step Required=true>
5.         <Name>1</Name>
6.         :
7.         </Step>
8.         <Milestone id=M2>
9.         <Step>
10.        <Name>2</Name>
11.        :
12.        </Step>
13.      </Flow>
14.    </Activity>
15.    :
16.    :
17.    <Activity id=A4 StartUseIf=M2>
18.      <Name></Name>
19.      <Flow>
20.      :
21.      :
22.      </Flow>
23.    </Activity>
24.  </Details>
```

Example 8: Triggering and Utilizing Milestones

In Example 8 milestone M2 is triggered between Steps 1 and 2 (see line 8). Milestone M2 is tested at the time Activity A4 is supposed to execute (line 17 - StartUseIf=M2). In effect, activity A4 will not execute unless activity A1 has executed at least once.

6.3 Activity Alternatives

As mentioned in Section 5, each activity has a primary execution scenario, and zero or more alternative scenarios. The different execution scenarios of an activity are expressed under the <Alternatives> tag depicted in line 25 of Example 6, and terminating on line 45. To express an activity alternative, we start by giving each activity alternative a unique "Id". Next we must specify under what condition the alternative is to be executed. For any set of primary and alternative activities, the condition for the primary execution scenario is evaluated first. If it succeeds, the alternatives are ignored. The RGML use milestones to express these conditions. The <StartUseIf> tag (Example 6, line 27) is used to identify the milestone that controls the execution of an alternative activity. Similarly, the <StopUseIf> tag is used to signify a milestone condition such that, when encountered, causes the current execution scenario is to terminate. If the <StopUseIf> tag is not used while defining an activity alternative, then if that activity is selected for execution, it will continue to be selected in subsequent passes through the flow until its milestone guard is reset.

All activity alternatives share the same name, goals and preconditions defined within the primary activity. The only difference between the primary activity and the

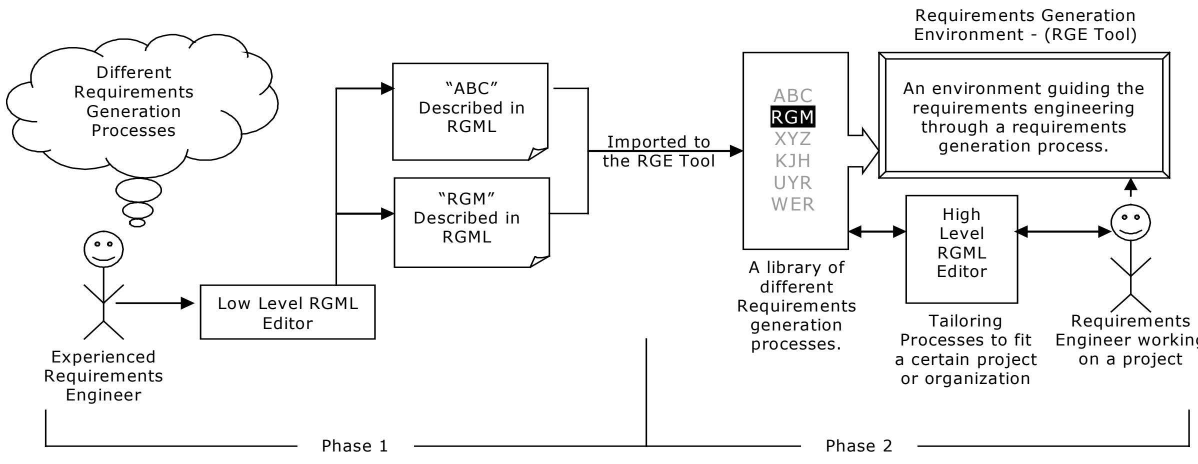
Different Requirements Generation Processes
Experienced Requirements Engineer
Low Level RGML Editor
"ABC" Described in RGML
"RGM" Described in RGML
Imported to the RGE Tool
ABC
RGM
XYZ
KJH
UYR
WER
A library of different Requirements generation processes.
Requirements Generation Environment - (RGE Tool)
An environment guiding the requirements engineering through a requirements generation process.
High Level RGML Editor
Tailoring Processes to fit a certain project or organization
Requirements Engineer working on a project
Phase 1
Phase 2

Figure 4: A Requirements Engineering Environment

alternative activities is what is defined within the `<Flow>` tags, i.e., the steps of the activity, the milestones to trigger, and when to trigger them.

7. Placing the RGML in Context

In the previous sections we have attempted to describe the principal capabilities of the RGML. More specifically, we have described how it is used to *formally* characterize

- the flow of control for a requirements generation process,
- the activities of such a processes, including the instantiation of applications and the creation of artifacts,
- temporal execution controls through the use of milestones and pre-conditions, and
- multiple execution scenarios for a given activity.

The remainder of this section briefly outlines our vision of an *Interactive Requirements Engineering Environment* and what role the RGML plays in realizing that environment.

Figure 4 illustrates a two-phased approach in defining an interactive requirements engineering environment that guides a requirements engineer through a structured requirements generation process. Phase one emphasizes the *definition* of requirements engineering processes. On the left side of the diagram we have an experienced requirements engineer who wants to formally characterize a requirements generation process. It may be the one currently being employed in the workplace or an experimental one. The requirements engineer employs a low-level structured editor that assists him/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, and
- the structured entry of RGML code directly.

As the RGML code is created, it is syntactically and semantically validated. When a complete requirements generation process is defined, its definition is stored in a database.

The second phase depicts two distinct activities – both are performed by the requirements engineer who is working with the customer to identify pertinent requirements. The first activity is process of identifying and recording customer requirements. This activity, however, is guided by an *interpretive tool* that

- reads 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.

The requirements engineer must also be given the flexibility to tailor a process to fit an organization's needs. Correspondingly, the second activity depicts the requirements engineer modifying the definition of the requirements generation process through the use of high-level editor. We envision the editor having 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 should permit tailoring during the requirements generation process or after it is completed. The editor should restrict process modifications should to those actions that do not violate fundamental principles underlying requirements generation. It should also be designed with the understanding that the person doing the tailoring may have only a minimal understanding of the RGML.

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

8. Conclusion

In this paper we have provided an outline of the Requirements Generation Markup Language. The RGML is a flexible language with significant expressive capabilities that support a formal characterization of the various processes defining requirements generation. The "structure" component of RGML permits the description of loops, branching, and sequence constructs. The "details" 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.

The RGML is novel and can be a benefit to the requirements engineering community in several ways. In being descriptive-based, initially the RGML will provide 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. Ultimately, we envision the development of an interactive, visually oriented environment that provides significant guidance and assistance leading to the elicitation, recording and packaging of a set of quality requirements. The RGML is the first step toward the realization of that environment.

9. References

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

[2] 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

[3] Craigh Schlenoff, Amy Knutilla, *and Steven Ray*, editors, Proceedings of the Process Specification Language (PSL) Roundtable, Gaithersburg MD, April 1997. National Institute of Standards and Technology.

[4] A. M. Davis. Software Requirements. Objects, *Functions and States*. Prentice-Hall, 2 edition, 1993

[5] 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.

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

[7] Herlea, D.E., Jonker, C.M., Treur, J., and Wijngaards, N.J.E. *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.

[8] Mike Marin. *Workflow Process Definition Interface---XML Process Definition Language*. The Workflow Management Coalition Specification, pp. 15-31,

[9] R.S. Pressman. Software Engineering, *A Practitioner 's Approach*. McGraw-Hill, Inc., 3rd edition, 1992.

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

[11] A. van Wijngaarden, *Orthogonal Design and Description of a Formal Language,* Mathematisch Centrum, MR 76, October 1965.

[12] Young, R.R. (2001), *Effective Requirements Practices,* Addison-Wesley Information Technology Series, Addison-Wesley, Boston Mass, 2001.

[13] Standish Group 1995. "Chaos", Standish Research Paper, <www.standishgroup.com/chaos.html>.