Process Definition and Communication

Frank Tsui
Department of Software Engineering, Southern Polytechnic State University, Marietta, Georgia, U.S.A.

Abstract
In today’s global economy and global approach to developing and supporting software, it is critical that the process utilized for such activities is clearly defined and communicated to all constituents. In this entry, the general topics related to software process, process definition, process rationale, process communication, and cost are first discussed. A specific list of items: activities, control, product and artifacts, resources, and tools (ACPRT) are introduced as categories of information that are necessary for a well-defined software process. Examples from graphic process languages such as Visual Process Language (VPL) and Little-JIL, from Ada-based programming-like process languages such as APPL/A (A Language for Software Process Programming) and CSPL (Concurrent Software Process Language), and from XML-based process definition and exchange languages such as XPDL and BPEL are then explored as instruments for conveying processes.

INTRODUCTION
Developing software and the subsequent modifications and updates of software involve complex and time-consuming tasks. Often, there are several alternative ways to perform these tasks. A well-defined process that is understood by all would certainly help software development and support projects. In this entry we first discuss what a software process is and what its definition should include. A differentiation between software process and software life cycle is then provided. The rationale and need for software process definition and communication is presented along with the cost that such activities may incur is explained. The notion that different perspectives that one may take, based on different goals, in defining the software process leads to potentially different definitions is introduced. A set of notations and techniques, ranging from graphical notations to programming languages, that may be used in representing software process and in communicating those process activities is then elucidated. Finally, samples from existing software process definition languages are discussed in terms of how much those definitions contain with respect to the earlier list of what the software process definition should include.

Software development and support process specifications define the tasks involved and describe how those tasks are to be performed. It is not the same as methodologies such as Object-Oriented or Component-Oriented development methodologies. A software process defines activities performed in a manner prescribed by a specific methodology, but it also includes other information related to the activities such as what resources are applied to accomplish those activities. In this entry we discuss two related topics: 1) software process definition and 2) how the process definition may be communicated with different notations and approaches.

The term software process is used in a general manner to include many types of processes used in a software project such as software development process, software repair and adaptation process, software upgrade and evolution process. Before discussing the topics of process definition and communication, it is important to provide some rationale behind the interest in these topics. Hence, we first discuss what a software process is and why it is necessary to have a software process definition. Software development does not occur in a vacuum. That is, the activities involved in software development are engaged for a reason, and the activities are often performed by different people with varying experiences. While not all software projects will use or need the same set of activities, many will share a common set. For the constituents of a software project to participate in the project in an efficient and professional manner, it is necessary for them to know and agree upon what set of activities will be performed by whom, in what order, with what methods and tools, and with what expectations.

A SOFTWARE PROCESS
There are many activities involved in the development and support of software. The most popular and most often recognized single activity is programming, the conversion of the designed solution into a representation in some programming language form, or coding, and compiling it. Programming also includes debugging and some level of testing of that program code. However, this single activity is not enough to develop or support a complex and large
software system. Programming alone is not sufficient for
the engineering of software. There are many other sup-
porting and necessary activities such as requirements
gathering and analysis, design and design documentation,
test case development and test case execution, integration
and build, training, installing. Each of these activities may
be further broken down into many more subactivities. For
example, within the design activity there may be a high-
level architectural design subactivity followed by a set of
low-level detail design subactivities related to database
tables, to user interface and screens, and to application
algorithms. Fig. 1 depicts the programming activity and
its breakdown into three subactivities of coding, compil-
ing, and unit testing. The dotted lines depict a looping
back with corrections in the event when compiling or
unit-testing subactivities detected some problem. No
other information such as resources, artifacts, or exit
criteria is included in Fig. 1.

Depending on the goal or goals of the software project,
different set of activities may be needed and emphasized.
Software development process is just a vehicle of carrying
out those activities. A process model is an abstract
representation of the actual process. As such, a process
model usually includes only part of the information related
to the actual process. The process model may be repres-
ented or specified differently, using different notations.
Here, we often use the terms software process model,
process definition, and process specification interchange-
ably. Software process specification includes the definition
of the activities and the description of how those activities
may be carried out. A software process specification is
mainly composed of the following two major components:

- The activities that are to be included for the software
  project
- The order in which these activities are to be performed

Software process specification may be further expanded
from the two major parts and refined to include the follow-
ing types of items:

1. Activities: detailed description of each of the
   activities
2. Control: the necessary precondition (entry criteria)
   and postcondition (exit criteria) for each activity and
   the ordering of the activities
3. Product & artifacts: the results and outputs from each
   of the activities
4. Resources: the people who should be performing the
   activities
5. Tools: the tools and techniques that are used to aid the
   performance of the activities

A software development and support process is composed
of many activities, is performed under a set of controlled
conditions of entry and exit criteria, may be performed by
multiple people with different skills, may require several
different outputs, and may use a variety of tools and tech-
niques. We refer to this set of process activities and related
items as the ACPRT (Activity, Control, Product, Resource,
and Tools) items. It is difficult and tedious to define all of
the above items. Often the software process specification,
for reasons such as providing flexibility or lessening rigid-
ity, places emphasis on one part and not the others. We
need all the above parts; however, the depth to which each
part must be defined may be different depending on the
goal. We may choose to define all five parts at a very high
level for a management overview or may choose to place
focus on the part that specifies detailed description of the
product results and the output artifacts for systems assur-
ance organizations to help in checking the results.

At a very detailed level, for the purpose of process
standards or task automation, we may focus on the activi-
ties, subactivities, control, and tools. In a sense, as Leon
Osterweil has observed,[1] specifying the software devel-
opment process may be viewed as similar to constructing a
software system itself, albeit a very complex one, which
carries out all the necessary tasks involved in developing
software. Vanderburg[2] also drew analogy between soft-
ware process and the software itself, using software struc-
tural characterization, such as coupling, to depict the
structural characteristics of Agile processes. Specifying
the process at the most detailed level would be equivalent
to performing the detail design and programming of the
software process itself.

Acuna et al.[3] defined software process to contain two
interrelated processes, the software production process and
the management process, which oversees the production
process. There are many elements from software produc-
tion process and from management process that need to be
considered. However, only a small set of vital elements,
similar to the list of ACPRT items, should be considered.
These are agents or actors, roles, activities, artifacts or
products, and events. Agents and roles are similar to the
resources and tools in ACPRT. Artifacts are the products in
ACPRT. Activity is the same as activity in the ACPRT list.
Events are like triggers for activities; thus they are similar
to the control in ACPRT.

In business enterprises, such as manufacturing, we actu-
ally have large commercial software such as SAP[4] that is
executed to help construct widgets. These complex and
large enterprise management software mimic and automate the manufacturing business process. Managing, defining, and specifying complex and large business process are being accomplished with a fairly high degree of success by commercial software packages such as CRM. These successes of commercial software packages have not only improved the business processes they have automated, but have also streamlined the overall business operations considerably. In the software industry, Rational Unified Process (RUP), which is now known as IBM RUP after IBM purchased Rational Software Corporation, has been widely adopted. There are four phases defined in RUP as follows:

1. Inception
2. Elaboration
3. Construction
4. Transition

The familiar activities of software requirements engineering, design, implementation, testing, and integration are mapped against these phases. RUP is a software process framework, rather than a single process, which has been used successfully by multiple software organizations. However, the software industry is relatively young and continues to evolve its process. There is still room for progress and further improvement of the process and the definition of process.

SOFTWARE LIFE CYCLE AND SOFTWARE PROCESS

Often times we hear the terms, software life cycle and software process, used interchangeably. It is worthwhile to take a moment and clarify the perspective differences or view differences between software life cycle and software development and support process. Both provide some characterization about the development and support of software. In the case of software life cycle we are addressing the states of the software. Software life cycle is a specification of a model of the software states. Software may start as just an idea. Then requirements are solicited and gathered, and software takes on the state of requirements notes. The software is then analyzed and documented into a requirements specification document. It moves from the state of “requirements notes” to “requirements specification.” As the software is designed it further changes state from architecture to detailed design. Software life cycle focuses on the representation and depiction of the software states as it evolves from inception to the final state when that software is withdrawn from service. With software life cycle, the view or the emphasis is not on the performance of activities but on the software entity itself.

In the case of software process, the focus is more on the representation of the actual activities performed on the software. The activities performed on the software will change the state, and allow us to move from the inception state to the desired final state. Software life cycle description, in many cases, should be included as a part of the broader software process definition. The IEEE Std 1074, IEEE Standard for Developing Software Life Cycle[7] is a standard for creating a software life cycle process (SLCP) that covers software development and maintenance. It begins with the selection of a software life cycle model (SLCM) first and then describes a set of activities to create the SCLP. This standard does not mandate any specific SCLM. It only defines the process by which an SCLP is produced.

NEED OF SOFTWARE PROCESS DEFINITION AND COMMUNICATION

Just as each software project is different and the project goals for each project may be different, the people involved in each software project may be different. Software engineers come from different backgrounds and experiences. They are equipped with different sets of skills and carry with them different biases. Without a defined and an agreed upon software process, there may be different assumptions among the participating software engineers as to which activities are to be performed in what order for the project, thereby creating great project turmoil. In addition, each activity may be carried to a different degree of depth with potentially different expectations on the form and content of the outputs. Each software project must include process definition and specification as part of its project planning activity and carry out the software project according to the planned process. The planning of project process does not imply that a huge, complex process is needed for the project. Even the deployment of simple process requires some level of planning. In the absence of a universally agreed upon single process for all software projects, we need to consider several factors before embarking on a specific process definition.[8]

- The specific nature of the project and its differences from other projects
- Unique needs and goals of the organization involved in the project
- The people, tools, product, et cetera, involved in the project

Besides the concern of successfully completing a software project, we are also interested in whether a process makes a difference in the final result, such as the quality of the product or the productivity of the participants in the project. Thus we need to be able to describe the process utilized and other information associated with that process.
Without a defined, planned, and monitored process, we cannot characterize what really occurred in the project, thus making a proper comparison of software projects difficult. In addition, there is always a demand for future improvement. To improve we must first understand what activities and how these activities were performed. Therefore, the software process must be well defined and monitored for the project. The benefits of software process definition may be summarized as below.\[9\]

- Facilitates understanding and communications of the activities to be performed
- Supports process improvement
- Supports project management
- Supports automation of process execution
- Supports process guidance

While a well-defined software process is important, that in itself is not enough to ensure project success unless the process is accepted and utilized by the whole organization. The introduction of a software process should be viewed in a similar fashion as an introduction of a new tool or a new methodology; it must be properly rolled out to the whole project team. There needs to be a period of assimilation after the communication and education of the process is done.\[10\]

The benefits of having a well-defined process and a well-trained team on that process do not come free. There is a price to software process definition and communication. That will be discussed next.

**COST OF SOFTWARE PROCESS DEFINITION AND COMMUNICATION**

It is interesting to note that there has been a growing recognition of the benefits of software process and process definition by Software Engineering Institute, SEI,\[11\] and International Organization of Standards (ISO).\[12\] SEI first introduced Capability Maturity Model (CMM) in early 1990s,\[13\] which is an assessment mechanism of organizational maturity by how much and how well the organization defines and performs the various processes related to software development. One of its main aims is to encourage organizations to continuously improve themselves through process improvements. In 2001, CMM was upgraded to Capability Maturity Model Integrated (CMMI),\[14\] which included more than just software development and management. There are multiple aspects to CMMI such as systems engineering, software engineering, integrated product and process development, supplier sourcing, and services. The International Standards Organization, jointly with International Electrotechnical Commission (IEC), has also played a key role in bringing process standards to organizations. One of the major worldwide standards for SLCP is the ISO/IEC 12207.\[15\]

This standard provides a structure of processes involving the following: 1) acquisition phase to initiate a project; 2) supply phase to plan a project; 3) development phase to design and develop the software; 4) operation phase to release and deliver the software to users; and 5) maintenance phase of software. However, it does not prescribe or dictate any particular life cycle model or software development method. In addition, there is a realization that software engineering is highly related to or may be viewed as a part of broader engineering of a total system, and ISO/IEC 15288,\[16\] a standard for systems life cycle process, is sometimes used as a companion to ISO/IEC 12207. This is similar, in concept, to the CMM and CMMI relationship.

Today, there is a worldwide realization and appreciation of the value of these various process standards and assessment efforts. This is especially true among those organizations who operate at the global level where multilanguage, multiculture, physical distance, personal trust, et cetera, all present a formidable challenge.\[17\] It is, however, not clear whether there is an equal realization or acceptance of the cost involved in first developing software process, defining the process, and then communicating it. The potential cost may include the following.\[9\]

- Time consumed in developing, constructing, and documenting process
- Recruiting and financing special skilled personnel
- Finance for short-term external experts and consultants

Furthermore, there is the additional cost to introduce, educate, and enforce proper usage of the defined process. The effort and time involved in rolling out a defined process and ensuring that the process is embraced and properly utilized cannot be underestimated, especially for a large internationally distributed organization whose members are usually drawn from broad and diverse backgrounds.

It should also be pointed out that this cost is not just a one-time expense, but is often a recurring expense. As an organization matures, as new technologies or techniques are invented, as new tools are developed, and as new personnel are introduced, some or all of the effort involved in defining, introducing, educating, et cetera, will need to be repeated. The recurring cost, which is usually less than the original, must be budgeted and made part of software projects’ planning exercise.

**DEFINING SOFTWARE PROCESS**

Earlier we have listed the five main ACPRT items that should be included in the specification of a software process. That speaks to the scope of the software process definition or specification. There are two additional perspectives to consider when defining a process. First, these items may be defined in different manners, using different...
notations. This perspective arises from the recognition that there is no single, universally accepted way to specify a software process and all of the subprocesses, at least not at the writing of this document.

Second, the ACPR items may not always be included in a definition. For example, some definition may only include the activities and their sequencing. This perspective arises from the recognition that organizations and projects may utilize a similar process, but may differ in the details. Or organizations utilize different processes simply because they have different project goals and have different levels of resources. Different situations would define the software process in different levels of detail with different points of emphasis. Additionally, there may be different reasons, based on different views, to define the process. The SPEARMINT project (Software Process Elicitation, Analysis, Review, and measurement in an INTEGRATED Modeling Environment project) for example, classified a number of process models by three views:

- Functional Perspective
- Behavioral Perspective
- Organizational Perspective

This is one less than the four perspectives listed by Curtis, Kellner, and Over, which includes an additional Informational Perspective. The Functional Perspective places emphasis on which process elements or activities are being performed and what information flows are relevant. The Behavioral Perspective places emphasis on the sequence in which the activities are performed. Organizational Perspective addresses the view of where and who in the organization performs the activities. The fourth perspective, Information Perspective, focuses on the information entities that are manipulated or produced by the activities.

Another approach is Humphrey’s proposal of three levels of Universal, Woldly, and Atomic process definitions, where Universal is the very high-level framework, Worldly level is deep enough for programmers and managers to use in projects, and the Atomic level provides the finer refinements for use such as for automation. One can also consider using just two levels. One level would describe the process in a form of high-level framework. The second would be a more detailed level of specification of the components that make up the framework. Humphrey has also viewed process from an individual activities’ time and quality management perspective in his Personal Software Process. In addition to the individuals, he recognizes that practically all large software projects are completed with a team of people and developed processes from a team-building and team activities perspective in his Team Software Process, while there are many different ways to categorize process definitions, we group the definitions and specifications of software process as follows:

- By notations utilized
- By levels of ACPR items captured

These two categorizations are not necessarily independent of each other. The notations we need to use may depend on the level or specific activity that is been captured and specified. Similarly, the depth of definition of that particular activity and information may dictate the type of notation that should be used. In viewing definitions of software process, regardless of its notation, form, or level of details, ultimately we must consider whether that definition serves its purpose or intended goal. This consideration of the purpose or goal first is much like the principle articulated by the “goal, question, metric” (GQM) paradigm. GQM itself is a process that may be applied to the measurement, study, and improvement of a well-defined process.

SOFTWARE PROCESS DEFINITION NOTATIONS AND TECHNIQUES

There is no one notation that can capture all the different aspects of defining a software process. The different types of ACPR items may require different ways to represent them. The notation is also influenced by the amount of these items one wishes to capture. To depict a software development process for a large, multiple-unit organization one may want to utilize the scheme of process in the large, as opposed to process in the small, and use protocols. The notations used at the higher, abstract level tend to be graphical in nature. As more details need to be specified, the notation turns more formal and more toward a textual language, natural or artificial.

The graphical notations may be as informal as drawing boxes and arrows with textual labels to give it some semantics. Depicting software process at the framework level, with labeled boxes and arrows, often only serves a limited purpose. In Fig. 2(A), there are three boxes to indicate that there are three distinct activities, and the two arrows indicate that activities B and C are to follow activity A. Even this very simple box and arrow notation diagram captures part

![Fig. 2](A) Simple box and arrow notation. (B) Overlaid box notation.
of the ACPRT aspects for defining a software process: the activity and the control. The details of the activities involved require more definition in some other form or notation. The timing of the sequence control also needs more than just the arrow. The tail and the head of the arrows are assumed to be read from the tail to the head and the implied control flow is from the tail to the head. Although both activities B and C are to follow after the completion of activity A, the simple diagram in Fig. 2(A) does not specify if they should start at the same time. Neither does it specify whether activity B or C may overlap with the tail end of activity A.

Another graphical way to depict the activity and control at the high level of a process is to use the notation of just boxes with the implied semantics that activities are to be performed sequentially from top to bottom and from left to right. Fig. 2(B) shows the same three activities as in Fig. 2(A). In addition, there are two more pieces of information implied with this notation.

First, while both activities B and C are to follow activity A, activity B may overlap with the tail end of activity A. Second, the size or length of the box C implies that some degree of either effort or time needed to perform activity C is more than that of either activity A or B. The additional information conveyed, through the size of and through the overlapping of the graphical figures, is not precise. In addition, there are other attributes related to process definition that we are interested in. For example, who should perform activity A or what condition must exist for activity B to overlap with activity A.

The input into and the output from an activity may also be represented in these graphical figures. One way is to adopt the Data Flow Diagram (DFD) approach\[26\] where the flow of data is represented by the label on the arrow between the boxes. This way the arrows depict not only the control flow but also the data flow. If more information is desired about the input and output data or artifacts, then an additional mechanism must be included to define those attributes.

Graphical notations such as Figs. 2(A), (B) are convenient and relatively easy to visualize, but they have many shortcomings and are not adequate enough for defining software process, except possibly at a high, framework level. They are well suited for describing the process in the large as opposed to process in the small, where much more details are defined. This does not mean that graphical notations cannot portray details of a process. They certainly can, but in doing so may diminish the visual simplicity.

Today, there is a well-established set of graphical modeling notations in software engineering, called the Unified Modeling Language (UML).\[27–29\] UML is a part of the modeling standard of Object Management Group.\[30\] UML diagrams have now evolved from 9 types of diagrams in UML 1.x to 13 types of diagrams in UML 2.0. These 13 types of diagrams are divided into three main categories:

- Structure Diagrams
- Behavior Diagrams
- Interaction Diagrams

The Structure Diagrams category is composed of six types of diagrams that represent static elements of a business or system structure. These are: 1) Class diagram; 2) Object diagram; 3) Component diagram; 4) Composite Structure diagram; 5) Package diagram; and 6) Deployment diagram. The second major category, Behavior Diagrams, includes three types of diagrams that depict behavioral features of a business or system process. The three types of Behavior Diagrams are: 1) Use Case diagram; 2) Activity diagram; and 3) State Machine diagram. The Interaction Diagrams category is similar to the general Behavioral Diagrams and emphasizes the interaction among the Objects. There are four types of diagrams in this category, and they are: 1) Sequence diagram; 2) Communication diagram; 3) Timing diagram; and 4) Interaction Overview diagram. This rich set of graphical diagrams, although intended for Object modeling, can certainly be applied to defining software processes. For example, the Activity diagram, which is used for depicting a high-level business process, can also be used for depicting high-level flow of activities of a software process. It also allows us to drill down to more detailed description of the activities and subactivities or nested subactivities of the software process. The Sequence diagram may be used to depict the sequential ordering of the activity flow and the related exchange of messages among the activities of the software process. The State Machine diagram can be used to show the software transition of states, or the software life cycle, as different software process activities are performed. An example of an Activity diagram is shown in Fig. 3.

---

**Fig. 3** Activity diagram depicting specification and review.
Develop requirements specification subactivity. The requirements specification activity, which is a part of the software process, is described at a high level in Fig. 3. The requirements specification activity is roughly composed of three major subactivities: develop the requirements specification, review the requirements specification, and approve the requirements specification. Each of these subactivities is represented with the rounded rectangle, called activity node. The diagram also utilizes arrows to show the sequencing of the subactivities; it also provides the feature of flow control with the diamond to alter the order. There is also an implicit flow of tokens through an activity diagram. These tokens are produced and consumed by the activity nodes and flow over the activity edges that are represented by the arrows. These tokens operate in a similar manner as those in a Petri net. If further refinement to this Activity diagram is needed, that can be easily accomplished. For example, the subactivity, develop the requirements specification, in the diagram may be further expanded to include three more, lower-level subactivities: 1) the requirements elicitation activity; 2) the requirements analysis and negotiation activity; and 3) requirements documentation activity. A new lower-level Activity diagram, labeled “Develop Requirements Specification”, with these three lower level of subactivities to further describe the develop requirements specification subactivity is shown in Fig. 4. Note that there is a rectangle in the lower right corner on the border of the Activity diagram. This rectangle represents the output parameter node. In this manner, Activity diagrams may be combined to drill down in the definition of a complex activity.

Different UML diagrams may be used together to define a process. In their entry describing global software process,[17] Vanzin and et al. used the Use Case diagram to describe the activity of software process owner and software process reviewers reviewing the global software process. The Use Case diagram was, however, accompanied with an extensive amount of English text to support the Use Case diagram. They further refined the description of this activity with an Activity diagram. Two different types of UML diagrams are used in a complementing fashion. There are many more features to the Activity diagram and other UML diagrams, but we do not pursue it here. Interested readers may check the references on UML at the end of this entry.

The tabular notation is another way to represent the software process. This is a hybrid between graphical and textual representation. The columns of the table may represent the major categories of process definition such as activity, resource, input, output, and tools. Each row of the table is then a separate activity with the definition of the associated information related to that activity.

Fig. 5 shows a portion of the software process in a tabular form. Coding activity A1 in Fig. 5 is a coding effort that takes the input, Design Spec, and converts it into a compiled code output, module x, using the compiler z tool. This coding activity is performed by J. Sims. Testing activity A2 is a testing effort that uses coded module x and test suite t as inputs and executes test suite t against the module x. The results from the testing effort are recorded in an output, test result. Test tool w is used in the performance of this activity, and the activity is performed by K. Rao.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Resource</th>
<th>Input</th>
<th>Output</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding</td>
<td>J. Sims</td>
<td>Design Spec</td>
<td>code module x</td>
<td>compiler z</td>
</tr>
<tr>
<td>Activity A1</td>
<td></td>
<td>code module x and test suite t</td>
<td>test result</td>
<td>test tool w</td>
</tr>
<tr>
<td>Testing</td>
<td>K. Rao</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activity A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5  Tabular notation.
Note that the assumed activity order is from top to bottom. But within a row, the text representation of information related to the activity may be read in any order. The amount of information placed in each row may also differ. The number of columns of the table may be expanded to accommodate more information such as the control flow information. For example, an additional column entitled “control flow” may be added. In that column one may specify the decision criteria for changing the assumed top-down sequential control flow and the activity to which the control should be given. The tabular notation provides a broad flexibility in that additional columns may be added and the amount of text is not restricted to some fixed number of words. The advantage of this notation is that it allows variations in breadth through additional columns, and depth through subtables and textual specification. One can use the tabular notation to define a software process at the high level and have separate subtables, using the same notation, to further refine the details of each of the major activity in the high-level table. The textual part of the table is usually some form of natural language. This advantage, however, may also be construed as a disadvantage because it allows great variability in the definition of software process. The loosely controlled text portion of the definition provides flexibility, but is also the source of potential ambiguity.

A more controlled manner of text representation in a form of some artificial language as a notation may offer a solution to the problem of ambiguity. A process definition language in the form of a formal language such as a programming language would be even less ambiguous. However, any programming-language-like representation comes with the complexity of a programming language and the associated learning curve. Imperative programming languages, as opposed to declarative languages, provide more details of how some process is to be performed. For higher level process definition, one may choose to use a declarative language type of approach where the focus will be on what the process is to do.

Defining a software process with an imperative programming-language-like approach allows us to describe the process in a very detailed manner. One can certainly adapt from existing programming languages such as C++ or Java and create a template of activities and the associated data that needs to be acted on with programming patterns to depict various activities in a software process. For example, one may “code” a requirements elicitation pattern using the programming language representation as a template for the activity of requirements elicitation. A programming language provides a spectrum of control structures of sequence, if-then-else, loop, case, et cetera, which allow one to depict the fundamental sequencing of activities in a process. Through data variables, it also allows one to depict the entry and exit criteria of the activities and other information such as the required resources or tools. Furthermore, if the process

programming language devised for the software process definition has a compiler and an execution environment, then we cannot only describe and enforce some rules of well-formed criteria on the activity definition, but even execute the process.

Aside from the programming language approach, various business consortiums have embraced the document markup language techniques to define different business workflow and business service processes. One can utilize a language such as Extensible Markup Language, XML, to depict the process. XML language is an information structuring language. In a sense, defining a software process is structuring a set of information pertaining to the software process. We can standardize the structure of information that describes a software process by using a schema language based on XML such as the Document Type Definition, or DTD, and develop a specific Software Process Markup Language for an organization. The major “elements” defined in the DTD may represent the key activities of the software process. The elements may also have attributes, such as performer and tool, to identify who would perform what activity with what tool. Also, an element may contain nested subelements to identify the sub-activities within a high-level activity. Such a Software Process Markup Language may be viewed as a template for describing the software process for a particular organization, and all software developers and service personnel in that organization have to agree to it and use it. One can also choose to use a more recent schema language, the XML Schema, which is an improvement over DTD. Just as XML, both DTD and XML Schema are proposals from the World Wide Web Consortium, or W3C. W3C has also recommended a sublanguage called XPath, which allows one to define a path of elements in an XML document. Using XPath, a sequence of activities may be identified in the previously mentioned Software Process Markup Language template. Thus, this sequence, which is a result of the XPath definition, may be read by a computer program and be “executed.” With an XML processor, a software process defined this way may be checked for its well-formed criteria and may also be executed.

Lastly, one may model the software process with a very formal language such as set theory notation or the Z language. The formal languages are concise and precise. However it is not practical in most commercial environments simply because of the amount of formalism involved, and the communications of the process defined with such a formal language notation will be too much a hurdle to cross for some of the constituents in a software project.

### SAMPLE PROCESS DEFINITION LANGUAGES

In this section, some of the existing process definition “languages” is discussed. Graphical languages using diagram approach is examined first. Then, process
programming languages with their roots in the Ada language is discussed. A process definition language that is closer to declarative language than imperative language is then presented. Finally, examples from the document markup language-based, or XML-based, process definitions proposed by the industry work groups will be introduced.

Visual Process Language (VPL)\textsuperscript{[31]} started as a language to prototype software processes. It is meant to visually represent the process and also permit the enactment of the software development process. VPL’s approach is to use a directed graph of nodes and edges. The rules for the graph are: 1) the graph must be fully connected and contains exactly one start node and one terminating node; 2) there are nine defined nodes, and only those may be used; 3) each input to a node is connected to only one of the output of another node; 4) there is no path from the output of any node back to its input except through a branch node. The nine defined nodes have graphical figures representing them as shown in Fig. 6.

These graphical-figure nodes are connected into a directed graph to represent and define the process of interest. The nodes may be labeled to further clarify the definition. The procedure node may have a name such as requirements gathering, and may, in turn, contain a connected graph of nodes, which expands on the tasks involved in requirements gathering. The task node, different from procedure node, represent a single atomic task. VPL is relatively easy to utilize and visualize. It covers the activities and the control of activities of a process definition. The definitions of artifacts or products, of resources, and of tools are not readily covered by this VPL graph. Separate tables will be required to define those. Thus, VPL covers items A and C of ACPRT.

Little-JIL\textsuperscript{[32,33]} is a language for coordinating process steps that are performed by agents who may be human or some automated entity. The syntax employed in Little-JIL is also graphical. It is aimed at easy visual understanding and adoption by its users. In Little-JIL, the basic element is a step. Each step is represented by a graphical icon of a rectangle with a set of “badges.” The badges represent information relevant to the step, such as how to handle exceptions or what are the pre- and postrequisites of a step. Fig. 7 shows an example of a step.

Each step’s behavior is defined via six states: posted, started, completed, terminated, retracted, and opted out. These states are self explanatory except for retracted and opted out. Retracted steps are those that are removed after being posted but prior to started state. These are usually steps that are part of a set of potential alternatives steps, but were not chosen to be started after one of the other alternative steps was chosen. Opted-out step is just a step that an agent chose not start.

Little-JIL uses a tree of these steps to represent a process. Fig. 8 shows a very simple tree structure.
The children of any step are those substeps that must be completed for that parent step to be considered as complete. The parent step to the children steps is represented by drawing a line from the parent step's sequence badge, which in Fig. 8 is the "═" sign, to the children steps. The order in which the children steps are performed in Little-JIL may be one of the following four choices:

- Sequentially: one at a time from left to right
- Parallel: where children steps may be performed concurrently
- Choice: where only one of the children steps is to be completed
- Try: performs the children steps from left to right until one of them is completed; if any child step terminates without completion, the next child step is tried

In Fig. 8, the parallel processing of substeps Def_Table1 and Def_Table2 is represented by the "═" sign in the sequence badge of the parent step, Def_DB. The interface badge of Def_DB declares that the agent who will perform the Def_DB step is DB_designer. How the leaf step, or atomic task, is actually performed is not specified by Little-JIL. Little-JIL only coordinates these steps.

The different badges represent information related to the step. Details of the information are written in text form just as the agent information was placed next to the interface badge in Fig. 8. The badges in Fig. 7 are defined as follows:

- Interface badge declares the parameter flow into the step, what resources it needs, and what exception the step throws.
- Handler badge indicates what exception the step handles.
- Requisite badges indicate the prerequisites and the postrequisites of a step. Various conditions, inputs, and outputs may be placed here.
- Sequence badge indicates the processing sequence of the children steps.
- Reactions badge indicates how a step dynamically responds to the arrival of an external message that signals the occurrence of an event.

Little-JIL is a fuller but more complex graphical language than VPL. Through the badges, it covers A and C of ACPRT. It addresses P,R, and T items of ACPRT mostly through labeling the badges, using text.

Aside from the pure graphical representation of process, there are some programming language-like process definition languages such as APPL/A [34] and CSPL [35]. Both of these languages are based on the Ada programming language and are fit for providing details of the process or the definition of process in the small. A salient feature of these process programming languages is that the "program" written in these languages may be compiled and executed, or enacted. One may view these languages as a programming language designed for a specific application domain, the software development and support process domain.

There is no room but to only illustrate some sample features from these languages here. We use examples from CSPL. Since the language is based on Ada, many of the object-oriented features such as inheritance are part of the language. For example, this allows a definition of a general software artifact class followed by a specific artifact class, such as a design document, to be inherited from the general artifact class. The language includes data variables and control structures for sequencing, branching, iteration, etc. just as a typical programming language. There are some interesting software process domain-specific features. One such feature is the support of assigning work to multiple developers using some specific tool. The syntax of the statement may look as follows:

```
3 testers edit testcase_doc referring to req_doc using editor;
```

This statement says that three testers are assigned to the task of editing the test case document called testcase_doc; the
three testers also refer to the requirements document called req_doc, and utilize the editor to accomplish this task. Other software process domain-specific features include the definition of tool unit, resource role unit, and relation unit. The syntax for the tool and role assignments may look as follows.

For a tool unit:

```java
tool dev_tool is
code_control = "CVS";
code_editor = "Jext";
compiler = "GCJ";
end;
```

This definition specifies a tool unit called dev_tool. The code_control variable is assigned the specific tool value called CVS, the code_editor variable is assigned the specific tool value, Jext, and the compiler variable is assigned the value of GCJ. Of course CVS, Jext, and GCJ are all specific names for the different tools. This coding syntax is very much like a typical programming language coding syntax.

For a role unit:

```java
role programmer is
  program_lead = "tjHernandez";
  programmer1 = "stThomas";
  programmer2 = "ckChen";
end;
```

The role definition for programmers specifies that there are program_lead, programmer1, and programmer2 variables. These are assigned the values tjHernandez, stThomas, and ckChen, which are real names of programmers in the organization. There are other software process domain-specific features such as asynchronous communications, but we limit our illustration here to just say that both APPL/A and CSPL, much like a programming language, require some time to learn and become proficient in. However, defining a software process with a process programming language allows us to capture the key features of a process definition: activities, resources, tools, products or artifacts, and the control conditions such as pre- and postconditions or entrance/exit criteria, all ACPRT items.

A slightly different process programming language is the Multi-View Process Language (MVP-L)\(^{[36]}\) first introduced as a part of a larger MVP modeling project on process modeling, on representation of process, on modularization based on views, and on the use of the models in the context of quality improvement at the University of Maryland. This language is based on three basic models: product_model, process_model, and resource_model. These three models are the fundamental building blocks of MVP-L. The product_model addresses the various artifacts and the final software deliverable. The process_model describes the various activities performed in a software project. Resource_model includes the resources and the tools needed to perform the activities that produce the artifacts. There are also three MVP-L attribute models that define the observable properties of each of the three basic MVP-L models. There is a fourth attribute model, called global_attribute_model, which is used to refer to general items such as time, date, or execution status. All user-defined models written in MVP-L are type description of, or derived from, one of these MVP-L model types. For example, users of MVP-L define an activity of the software process in the form of user process_model, which contains clauses that tie together what was defined in the user resource_model and user product_model. It further contains clauses that define what attributes need to be imported and what need to be exported. It describes the activity in terms of what is consumed and produced, not how consumed and produced. It also lists the entry and exit criteria for the activity. This is a bit like a rule-based programming or a logic programming language. The MVP-L models capture the software process activities, the controls in terms of entry and exit criteria, the product and artifacts, the resources, and the tools used to perform the activities. This language also fully captures all of the ACPRT.

We now turn to a considerably different process definition approach. This is an approach that views the definition of a process as a document that describes the process. The approach is based on utilizing a document markup language such as XML. Different professional business organizations have proposed several business flow and business process languages such as XML Process Definition Language, XPDL\(^{[37]}\) and Business Process Execution Language, BPEL.\(^{[38]}\)

XPDL is a standard language proposed by the Workflow Management Coalition, WfMC. It is a widely deployed process definition language whose implementation includes companies such as Adobe, IBM, BEA, and Software AG. It is a process definition language that models a business workflow. The resulting “diagram” provides a format for interchange and communication of the definition. Tools are needed to edit the definition or to “execute” the definition. The XML file format provided by XPDL can be used as a standard to exchange process models between tools of process model building and tools of process model run-time interpreting and execution. These tools may be provided by multiple and different vendors. Vendors need to be able to import and export XPDL formatted files. The vendor’s internal file format for processing the XPDL files is not of concern and may be anything the vendor chooses.

To facilitate the interchange of workflow definitions, a workflow process definition Meta-Model has been established to identify commonly used entities within a process definition. The claim of conformance to the Meta-Model
by vendors is authorized by WfMC, who verifies the claim through defined test procedures over a minimum set of mandatory objects. The General XPDL Meta-Model is shown in Fig. 9. The Workflow Process Definition shown at the top of Fig. 9 is a container for all the elements below it, which include: 1) a network of activities and their relationships (e.g., Activity Set, Workflow Process Activity, Block Activity, Atomic Activity); 2) criteria to indicate the start and termination of the processes (e.g., Transition Information); and 3) information about the individual activity such as participants (e.g., Workflow Participant Specifications), associated tools, and data used in process during process execution (e.g., Workflow Relevant Data). In addition the process definition may reference a subactivity flow (e.g., SubProcess Definition), which is separately defined but together make up the overall flow.

There are defined Process Meta_Model and Package Meta_Model. The Process Meta-Model, for example, contains the basic elements (entities and attributes) that must be defined for the exchange of process definition and includes the following items from the General XPDL Meta-Model:

- Workflow Process Activity
- Transition Information
- Workflow Participant Specification
- Workflow Application Declaration
- Workflow Relevant Data

A sample of XML Schema code depicting part of the Activity Set in the Workflow Process Definition in Fig. 9 is shown below to illustrate how XPDL interchange is defined.

```xml
<xsd:element name = "ActivitySet">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref = "xpdl:Activities" minOccurs = "0"/>
      <xsd:element ref = "xpdl:Transitions" minOccurs = "0"/>
    </xsd:sequence>
    <xsd:attribute name = "Id" type = "xsd:NMTOKEN" use = "required"/>
  </xsd:complexType>
</xsd:element>
```

Here an Activity Set is defined to include Activities and Transitions in that order. The Transitions in the Activity Set should only address those activities within the set and not into or out of the set. Also, the attribute declaration indicates what the data type is and that this attribute is required.

We will not expound on BPEL except to say that it is also an XML-based language that many business analysts use to define and coordinate web services and the operational details. It is a web service standard developed by the Organization for the Advancement of Structured Information Standards, OASIS. BPEL is considered an “orchestration” language in that it focuses on the flow and control of the services as opposed to the “choreography” language, which focuses on the interaction and exchanges of messages among the service parties. Both XPDL and BPEL cover all of the ACPRT items for process definition.
CONCLUSION

Software process definition and the communication of that process has been a difficult and elusive subject. In today’s global economy and global approach to software development and support, it is even more important that a software organization or a group of partner software organizations define and communicate their software process as clearly and as effectively as possible.

In this entry the notion of defining a software process and what that definition should include was first explained. One specific list of items for the definition, which included 1) activity; 2) control; 3) product and artifacts; 4) resources; and 5) tools in the form of the acronym, ACPRT, was introduced. A special excursion was taken to explain the difference in perspectives between the term software life cycle and software process. We also point out that while there is a considerable amount of compelling reasons for defining and communicating software process along with the establishment of process standards, the fact still remains that these efforts come with a hefty cost.

Various software process definition notations and techniques, ranging from graphical to process programming languages, were discussed in an abstract manner first. Then samples of very specific process definition languages were introduced and expounded in terms of the level of inclusion of ACPRT items. VPL and Little-JIL were the two graphical process languages discussed. These were both strong in A and C parts of ACPRT, with textual ways to express P, R, and T. Two Ada Programming language-based process definition languages, APPL/A and CSPL, were introduced along with a slightly different multi-view process-oriented MVP-L language. These process definition languages are capable of capturing all the details, or processes in the small, of a software process. These languages can include the complete ACPRT. Finally, examples from business workflow process definitions using XML Schema Definitions were shown. Both XPDL and BPEL, which define popular, standard business workflow interchange formats, are capable of capturing all of ACPRT.

In light of the success that e-commerce is enjoying with standard business formats and business workflow using XPDL and BPEL, it would seem reasonable for software engineers to explore the same potential. Software industry is no different from other industries, with various tasks outsourced and processed around the globe. Software processes will no doubt continue to evolve and improve. However, the definition and communication of the software process can stand some standardization among the constituents who are partners in developing or supporting the software.

REFERENCES

10. Tsui, F. Managing Software Projects; Jones and Bartlett: Sudbury, MA, USA; 2004.


