ELSEVIER

Information and Software Technology 40 (1998) 311–325

INFORMATION AND SOFTWARE TECHNOLOGY

Enacting object-oriented methods by a process environment

Jen-Yen Jason Chen[a,*], Shih-Chien Chou[b]

[a]*Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan*
[b]*Department of Information Management, Ming Hsin Institute of Technology, Hsinfong, Taiwan*

Received 22 July 1997; received in revised form 16 March 1998; accepted 18 March 1998

Abstract

This paper describes modeling and enactment of two object-oriented methods, namely the OMT method and the Booch method, using the concurrent software process language (CSPL). CSPL is a process-centered environment using Ada95-like syntax to model a method as a process program which is then compiled to C-shell executing in UNIX. A method can thus be enacted (executed) using this approach. Experiences of the approach are depicted. It is demonstrated that CSPL is feasible and appropriate for modeling and enacting methods, including object-oriented methods. Additionally, partial CSPL program modeling the Booch method is included in the Appendix. © 1998 Elsevier Science B.V.

Keywords: Software process; Process-centered environment; Process modeling

1. Introduction

Large software development needs collaboration of distributed developers working concurrently. A software process environment is thus needed in assisting developers who are distributed geographically. The concurrent software process language (CSPL) [1]–[3] and its associated process-centered environment are developed attempting to satisfy this need. CSPL assists communication between developers and managers in a structured and controlled way. In addition, it manages software objects (documents), and it binds tools.

Fig. 1 gives a sketch of the CSPL environment that adopts a client–server architecture. The CSPL server is the kernel of the environment. According to the execution of a CSPL program (to be described), the CSPL server assigns work to the right developer at the right time. In this environment, each developer uses a CSPL client. Through CSPL clients, developers receive work assignments, perform work, access objects (documents), send the work result back to the server, and so on. Also, objects are stored and managed by the CSPL object management system (CSPL/OMS).

Two popular object-oriented methods are modeled in this paper, namely the Booch method [4] and OMT (object modeling technique) [5], using CSPL. Fig. 2 sketches the CSPL modeling and enactment approach. The modeling produces CSPL process programs, which are then compiled to C-shell scripts by the CSPL compiler. Next, the C-shell scripts are enacted (executed) on the UNIX system by the CSPL server.

Two motivations are noted for this research.

To evaluate the object-oriented method

There are quite a few object-oriented methods available. Practically speaking, what are their performances in terms of improving software quality and development productivity? As previously mentioned, a method is modeled as a CSPL process program. The process program can then be enacted to develop software. During the enactment software metrics are collected by which performance of the method can be evaluated to a certain extent.

To evaluate procedural process language

CSPL is an Ada95-like procedural process language. Experiences of using CSPL in modeling object-oriented methods allow us to evaluate this procedural process language approach. Meanwhile, other approaches such as Weaver [6], Hakoniwa [7], EPOS [8], and so on, are discussed.

This paper is organized as follows. Section 2 gives an overview of CSPL. Section 3 describes the modeling and enactment of two object-oriented methods using CSPL.

* Corresponding author. Tel.: +886 35 713928; fax: +886 35 724176; e-mail: jychen@csie.nctu.edu.tw

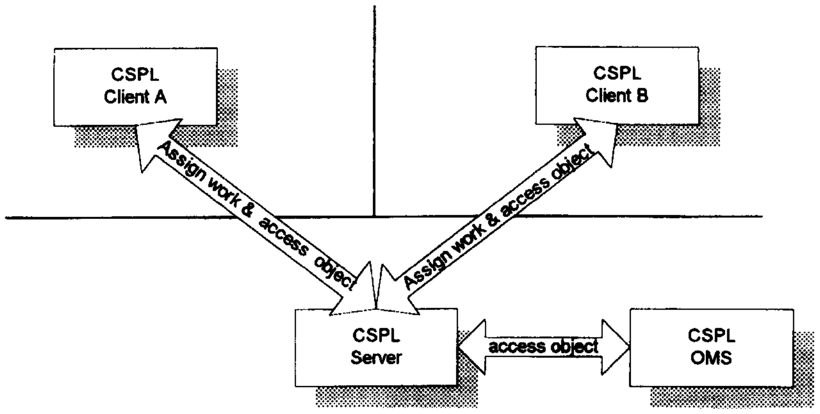


Fig. 1. CSPL environment architecture.

Section 4 depicts experiences gained. Section 5 briefly discusses some related work. Finally, Section 6 gives the conclusions and future work.

2. CSPL overview

A CSPL program looks similar to an Ada95 program. It contains procedures, functions, tasks, packages, generic units, exception handlers, and so on. To meet the special needs of a software process, new CSPL constructs are designed, such as tools, roles, metrics, and so on. The CSPL program structure is given next, followed by the description of some constructs.

2.1. CSPL program structure

The general structure of CSPL programs is illustrated in Fig. 3. Examples below are extracted from the CSPL program in the Appendix that models the Booch method.

At the beginning of a CSPL program, tools and roles are defined (see Example 1). The correct tool can be automatically invoked when used by a developer in an activity. For instance, in Example 1, "vi" is used as the editor. Also, a developer can locally define his or her preferred tool from a CSPL client. Note that in CSPL a developer can play multiple roles, and multiple developers can assume a role. For instance, in Example 1, "ymchen" plays three roles, namely "requireAnalyst", "RAreviewer", and "DomainAnalyst".

CSPL packages encapsulate program units such as procedures and functions. A package consists of two parts: package specification and package body. The package specification in Example 2 defines names and parameters of program units and object types accessed. The contents of a program unit are described in its body rather than in its specification.

It is worth noting that CSPL is strong in object modeling in areas such as object type, inheritance, and so on. In Example 2, object type "*DomainAnalysisDocument*" is derived from object type "*Document*". To meet the needs of the software process, CSPL currently provides built-in object types such as "*DocType*", "*TextType*", and "*NonTextType*".

In the package body in Example 3, the work assignment statement:

1 DomainAnalyst edit domainAnalysisDocument referring to

requireDocument using CASETool;assigns the work of editing "domainAnalysisDocument" to the role "DomainAnalyst" (which consists of one or more developers; see Example 1). The developer can refer

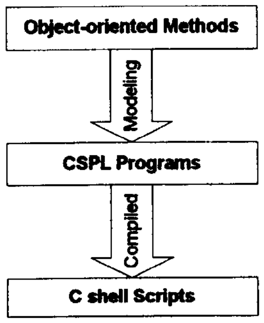


Fig. 2. CSPL process modeling and enactment.

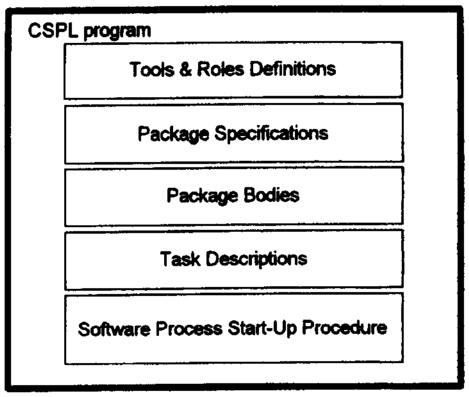


Fig. 3. CSPL program structure.

```
-- Tool Definition --

tool Booch_tools is
Editor := "vi";
WordProcessor := "interleaf";
CASETool := "ROSE";
ReviewTool := "reviewTool";
MetricTool := "OOmetric";
end;

-- Role Definition --

role requireAnalyst is
requireAnalyst1 := "ymchen";
end;

role RAreviewer is
RAreviewer1 := "jychen";
RAreviewer2 := "hsubj";
RAreviewer3 := "ymchen";
end;

role DomainAnalyst is
DomainAnalyst1:="ymchen";
DomainAnalyst2:="hsubj";
end;
..........
end;
```

Example 1. Tools and roles definitions.

to "requireDocument" and use the CASE tool. Although work can be directly assigned to a developer (such as "DomainAnalyst1"), it is suggested that work is assigned to a role instead. In this case CSPL can take into account the workload at enactment time, and accordingly assigns work to a developer of that role.

Example 4 depicts nested packages where a package contains multiple smaller packages. In Example 4, the package "Booch_Method" contains the packages "Problem_Statement", "RequirementAnalysisPhase", "DomainAnalysisPhase", "Design_Phase", "Evolution-Phase", and "MaintenancePhase". This construct encourages developing modular, thus understandable and maintainable, programs.

Like other program units, a CSPL task consists of a specification and a body. A task consists of procedural calls, task entry calls, and so on to model the flow of activities. In Example 5, task "DomainAnalysis" includes a procedural call (DomainAnalysisPhase.DomainAnalysis), a function call (DomainAnalysisPhase.ReviewDA), an exception handler (timeout), and some entry calls (such as Design.start and RequirementAnalysis.start). Also, multiple CSPL tasks run concurrently in a software process.

2.2. Synchronous and asynchronous task communications

CSPL supports two task communication modes—*entry call* and *event inform* (see Fig. 4). "Entry call" synchronizes tasks. The semantics of Ada rendezvous is preserved in CSPL. In Fig. 4, task A issues an entry call "B.start" to task B. On the other hand, "event inform" relates to asynchronous communications. In Fig. 4, after task C informs task D to set event OK, task C proceeds execution without waiting for task D. While the "waitfor OK" statement in task D waits until the event is set.

2.3. Relation between objects

A CSPL relation unit defines the relationship between two object types. Operations such as "insert" and "delete" are used to manipulate entries of relation units. Assume that an entry of a relation unit defines that document B depends on document A. When document A is updated, the relationship

```
-- Package Specification for Domain Analysis Phase

package DomainAnalysisPhase is

   type DomainAnalysisDocument is new Document with record
        classDiagram                : NonTextType;
        objectScenarioDiagram       : NonTextType;
        dataDictionary              : TextType;
        classSpecification          : TextType;
             workpartition          : TextType;
   end record;

   procedure DomainAnalysis(
         domainAnalysisDocument: in out DomainAnalysisDocument;
         requireDocument: in RequireDocument);
   function ReviewDA(
                domainAnalysisDocument: in DomainAnalysisDocument;
        requireDocument: in RequireDocument)
        return integer;

end DomainAnalysisPhase;
```

Example 2. Package specification.

```
-- Package Body for Domain Analysis Phase --

package body DomainAnalysisPhase is

   procedure DomainAnalysis(
         domainAnalysisDocument: in out DomainAnalysisDocument;
         requireDocument: in RequireDocument)
    is
      begin
                  1 DomainAnalyst edit domainAnalysisDocument referring to
                                 requireDocument using CASETool;
      end;

  function ReviewDA(
                  domainAnalysisDocument: in DomainAnalysisDocument;
        requireDocument: in RequireDocument)
        return integer
     is
       current_time : time;
       DomainAnalysisDeadLine : time;
       timeout : exception;
           workresult : integer;
           begin
         current_time := GetCurTime;
         if current_time > DomainAnalysisDeadLine
           then raise timeout;
         end if;

         all DAreviewer review domainAnalysisDocument referring to
            requireDocument using ReviewTool resulted in workresult;
         return workresult;
       end;

end DomainAnalysisPhase;
```

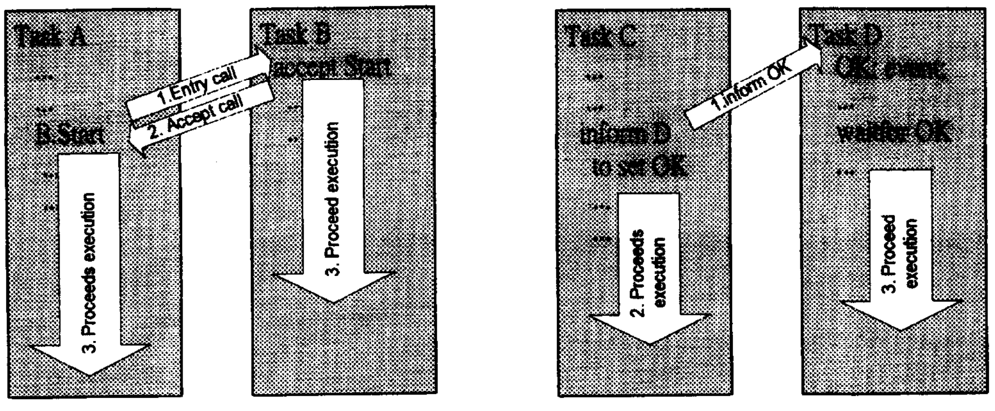
Example 3. Package body.

unit will trigger procedures to update document B. For example, object code is dependent on source code. Thus, if a source code is changed, a procedure should be triggered to compile the new source code and generate a new object code.

2.4. Exception handling

Exception handling is important in software process modeling because of abundant unexpected and irregular conditions in processes such as requirement change, schedule delay, and so on. In order to manage software processes, a process language needs to model those conditions as exceptions and specify their handling procedures.

In handling an exception a project manager sometimes needs to set (or reset) events. A menu listing all the exceptions and events of a CSPL program is provided by the CSPL user interface system for the manager to set the events.



(a) Entry call (b) Event inform

Fig. 4. Synchronous and asynchronous communications.

```
package Booch_Method is
   package Problem_Statement is
    .......................
    end Problem_Statement;
   package RequirementAnalysisPhase is
    .......................
    end RequirementAnalysisPhase;

   package DomainAnalysisPhase is
    ....................
    end DomainAnalysisPhase;

   package Design_Phase is
    .......................
    end Design_Phase;

   package EvolutionPhase is
    ........................
   end EvolutionPhase;

   package MaintenancePhase is
    .........................
   end MaintenancePhase;
end Booch_Method;
```

Example 4. Nested packages.

2.5. Review

The CSPL review statement originally allows only a "yes" or "no" decision from a reviewer. However, in modeling software processes, it is found that this leaves too little choice for the reviewer, which results in over-simplistic models that fail to meet practical needs. For instance, after a reviewer reviews a document in phase B, he/she may need to have the following choices: (1) the document quality is OK, proceed to phase C; (2) the quality is not OK. Rework of phase B is needed; (3) the quality is not OK, which results from poor quality in phase A. Rework of phase A is thus needed. Go to phase A.

The review statement is revised accordingly. In the statement shown below, all system reviewers review the system design document referring to "q.doc" document:

all system_reviewer review system_design_doc referring to q.doc *options*

analysis_review_option using review_tools resulted in review_result;

```
-- Task specification of Domain Analysis
task DomainAnalysis is
    entry start;
 end;

-- Task body of Domain Analysis

task body DomainAnalysis is
  begin
   loop
    accept start;
    loop

          DomainAnalysisPhase.DomainAnalysis(
            domainAnalysisDocument,requireDocument);
    DA_review_result := DomainAnalysisPhase.ReviewDA(
      domainAnalysisDocument,requireDocument);

    exit when DA_review_result > 0;
    end loop;

    if  DA_review_result = 1 then
       output "************************************************************";
       output "******** Reviewers Decide to Advance to Design Phase ******";
       output "************************************************************";
       Design.start;
    end if;

    if DA_review_result = 2 then
       output "************************************************************";
       output "**** Reviewers Decide to Back to Requirement Analysis ***";
       output "************************************************************";
       RequireAnalysis.start;
    end if;

   end loop;

-- Exception Handling for Time Out --
  exception
   when timeout =>
      output "Time out! Starting Design Phase";
      Design.start;
end;
```

Example 5. Task specification and body.

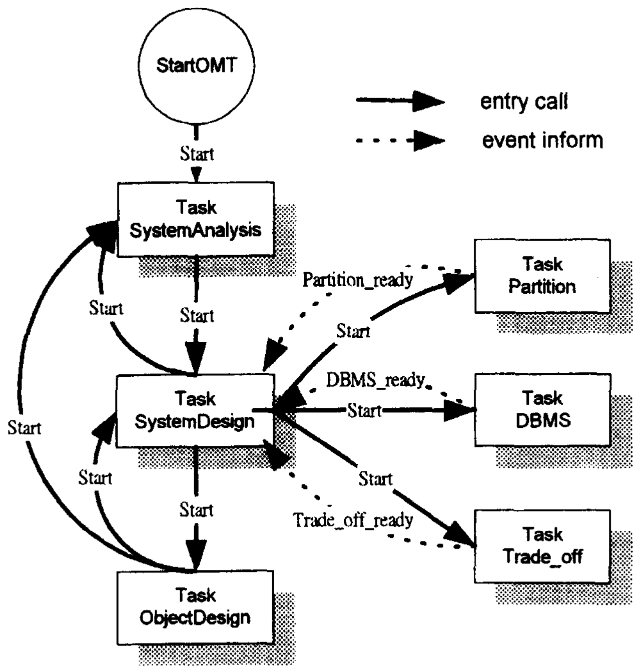


Fig. 5. OMT process model.

Variable "analysis_review_option" following keyword "options" contains all the choices for a reviewer. For instance:

ananlysis_review_option: =

" %1.OK,Go to Phase C %2.Redo Phase B %3. Go to Phase A ";

A window shows all the review options for each reviewer. After on-line discussions, reviewers hopefully will reach a decision (a consensus or majority vote) which is saved in variable "return_result". That variable apparently will determine the flow of the software process.

2.6. Metrics

Metrics are constantly used in evaluating analysis and design results to assure software quality. Code metrics such as lines of code have been available for a long time. When a process enacts an object-oriented method, object-oriented design metrics [9] are needed. In a CSPL program, when an object is reviewed, its metrics value will be passed to reviewers. Quality decisions can be made by referring to the metrics. This is expected to reduce implementation difficulties and development risks [10].

The following statement is used for that purpose:

measure Design_Document using DD_Metric_tool;

"*Design_Document*" above is the object being measured. "*DD_Metric_tool*" is the metrics tool for the object. The measurement result is stored in the attribute *metricResult* of "*Design_Document*".

Let us emphasize that metrics statements can be very effective in evaluating a method, including an object-oriented method—provided they are properly modeled and enacted.

3. Modeling and enactment

Section 3.1 depicts the modeling steps. The steps are applied to the OMT method and the Booch method in Section 3.2, Section 3.3, respectively. As a result, CSPL programs are developed. The CSPL programs are then enacted (executed) using the CSPL environment, as depicted in Section 3.4.

3.1. Modeling steps

The object-oriented methods are modeled in the following steps.

3.1.1. Step 1. Find activities and object types

The activities and object types in a software process can usually be found in the definition of the object-oriented method. They can be found for OMT [5] and for the Booch method [4], respectively.

3.1.2. Step 2. Merge cohesive activities

If all the activities found in step 1 are modeled as separate subprograms, the model is likely to be unnecessarily complex. To reduce this complexity, activities that frequently interact with each other are merged into one *coarse-grained activity* (see the discussion in Section 4.1).

Although the merging can reduce complexity, it may reduce the scope for multiple developers to work concurrently. Therefore, when modeling a process, one should take into consideration the trade-off of merging or not merging. Generally, the following activities are candidates for merging: activities assigned to the same developer, activities that are not concurrent, activities that take short time, and so on.

3.1.3. Step 3. Identify concurrent tasks

Related coarse-grained activities in step 2 are grouped into one module, which is then modeled as a task. Activities in different tasks are expected to be executed concurrently. Thus, if two activities are supposed to run concurrently, they must be allocated in different modules.

3.1.4. Step 4. Build connections between tasks

As just mentioned, step 3 constructs tasks. To coordinate and synchronize the tasks, connections between tasks must be built using the *entry call* and *event inform* constructs. Normally an entry call is used to synchronize or start tasks; while an event inform is used to set an event.

3.1.5. Step 5. Find exceptions

Exceptions and their handling procedures are used to model unexpected conditions. Exceptions are method independent, but their handlers are not. That is, exceptions for different methods appear similar. For example, exception "requirement change" may happen in the OMT or Booch method. However, different handling procedures are used in different methods. For instance, when exception "requirement change" occurs, the Booch method dictates a software process to backtrack to phase "requirement analysis", while OMT requires the process to backtrack to phase "system analysis".

3.2. Object modeling technique (OMT) method

OMT (object modeling technique) is a popular object-oriented method. It contains three phases: (1) system analysis; (2) system design; and (3) object design. Steps in Section 3.1 are applied to model OMT as described in the following.

3.2.1. Step 1. Find activities and objects

The activities and object types used in OMT are defined in Ref. [5] such as *Identifying classes*, *Identifying operations*, *Identifying relations*, *identifying concurrency*, *object model*, *dynamic model*, *functional model*, and so on.

First, object type "*Document*" is defined which contains attributes such as *reviewOption*, *reviewResult*, *reviewComment*, *authorComment* and *metricResult*. Then, the object types [5] such as *AnalysisObjectModel*, *AnalysisDynamicModel*, *AnalysisFunctionalModel*, and so on are derived from "*Document*".

3.2.2. Step 2. Merge cohesive activities

After the merge, activities and object types are listed as in Table 1.

3.2.3. Step 3. Identify concurrent tasks

The related activities are grouped into tasks. After grouping, six concurrent tasks are constructed: (1) *SystemAnalysis*; (2) *SystemDesign*; (3) *ObjectDesign*; (4) *Partition*; (5) *DBMS*; and (6) *Trade_off*. Among them, the task *SystemAnalysis* and *ObjectDesign* are for the system analysis and object design phases, respectively. They are composed of the activities belonging to the phases. The tasks *SystemDesign*, *Partition*, *DBMS*, and *Trade_off* are for the system design phase. The task *SystemDesign* is composed of these activities: *IdentifyConcurrency*, *AllocateSubsystem*, *ManageGlobalResource*, *ManageSoftwareControl*, *HandleBoundaryCondition*, and *ReviewSystemDesign*. The task *Partition* is composed of the activity *DivideSubsystem*. The task *DBMS* is composed of the activity *ChooseDBMS*. The task *Trade_off* is composed of the activity *HandleTradeOffs*.

3.2.4. Step 4. Build task connection

Fig. 5 shows entry calls and event informs among the OMT tasks. As mentioned earlier, entry calls are used to start other tasks, and event informs are used to inform other tasks about the occurrence of some events.

3.2.5. Step 5. Find exceptions

Three exceptions, namely (1) schedule delay, (2) requirement change, and (3) project cancel are found in OMT. Exceptions can be raised either by the CSPL program or by the manager. For instance, exception "schedule delay" can be raised by a raise statement in the program. Exceptions "requirement change" and "project cancel" can be raised interactively by the manager during enactment.

Exception handler for "RequirementChange" is shown below:

```
exception
when RequirementChange = >
output "Requirement Change!!";
output " REDO software.";
SystemAnalysis.start;
```

If this exception is raised, the handling procedure shows messages "Requirement Change!!" and "REDO software." at the CSPL server, and then issues an entry call to task "SystemAnalysis" to start redoing the software.

3.3. Booch method

The Booch method [4] is another popular object-oriented

Table 1
Activities and object types for OMT

	Activities	Object types
System analysis phase	BuildObjectModel	AnalysisObjectModel
	ReviewObjectModel	
	Build DynamicModel	AnalysisDynamicModel
	ReviewDynamicModel	
	BuildFunctionalModel	AnalysisFunctional Model
	ReviewFunctionalModel	
System design phase	DivideSubsystem	SystemDesignDocument
	IdentifyConcurrency	
	AllocateSubsystem	
	ChooseDBMS	
	ManageGlobalResource	
	ManageSoftwareControl	
	HandleBoundaryCondition	
	HandleTradeOffs	
	ReviewSystemDesign	
Object design phase	FindOperation	DesigningObjectModel
	DesignOpAlgorithm	
	OptimizeAccessPath	
	ImplementSoftwareControl	
	ModifyClassStructure	
	DesignAssociation	
	ClassRepresentation	
	ClassModulation	
	ReviewObjectDesign	

Table 2
Activities and object types used in the Booch method

	Activities	Object types
Requirement analysis phase	RequirementAnalysis	RequireDocument
	ReviewRA	
Domain analysis phase	DomainAnalysis	DomainAnalysisDocument
	ReviewDA	
Design phase	Design	DesignDocument
	ReviewDesign	
Evolution phase	Evolve	RequireDocument
	ReviewEvolution	DomainAnalysisDocument
		DesignDocument
		Program
		ObjectCode
Maintenance phase	Maintain	RequireDocument
	ReviewMaintenance	DomainAnalysisDocument
		DesignDocument
		Program
		ObjectCode

method. It has five phases: (1) requirement analysis; (2) domain analysis; (3) design; (4) evolution; and (5) maintenance. The process for the Booch method is modeled as described in the following steps.

3.3.1. Step 1. Find activities and objects

The activities and object types used in the Booch method are defined in Ref. [4]. The activities include those in the micro development process (i.e. the activities *identify classes and objects*, *identify class and object semantics*, *identify class and object relationships*, and *specify class and object interface and implementation*) and other activities, such as *architecture design*, *class diagram*, *object diagram*, and so on. Object type *Document* defined in OMT in Section 4.2 is reused here.

3.3.2. Step 2. Merge cohesive activities

After the merge, activities and object types are listed as in Table 2.

3.3.3. Step 3. Identify concurrent tasks

After grouping the activities, five concurrent tasks are constructed: (1) *RequireAnalysis*; (2) *DomainAnalysis*; (3) *Design*; (4) *Evolve*; and (5) *Maintain*. Each is for a phase and is composed of the activities in the phase. For example, the task *DomainAnalysis* is for the domain analysis phase and is composed of the following activities, namely *DomainAnalysis* and *ReviewDA*.

3.3.4. Step 4. Build task connection

The communication among tasks is shown in Fig. 6. Note that the tasks *Evolve* and *Maintain* are, respectively, initiated by the events *evolution requirement* and *maintenance requirement*, which are set by the manager. As mentioned earlier, entry calls are used to start other tasks, and event informs are used to inform other tasks about the occurrence of some events.

3.3.5. Step 5. Find exceptions

Exception handling of the Booch method is similar to that of the OMT method discussed in Section 3.2.

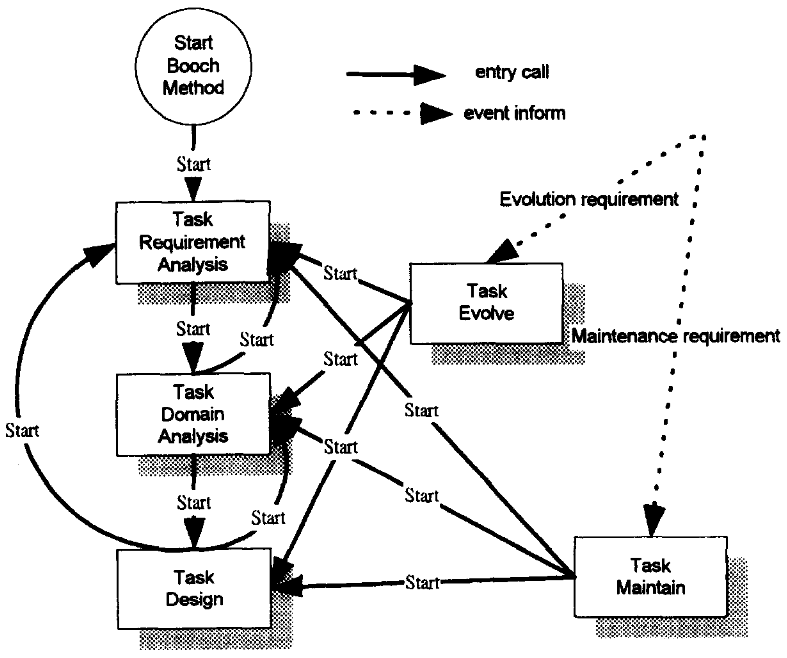


Fig. 6. Booch process model.

3.4. Enactment

After the object-oriented methods are modeled as described above, the resulting CSPL programs are enacted in the CSPL environment.

Tools used during enactment should be defined in the tool unit of the CSPL program. For example, to use a tool for the Booch method, the tool ROSE developed by Rational [11], [12] or OOCASE developed by the Institute for Information Industry [13] can be defined in that unit. On the other hand, to use a tool for OMT, the tool OOAid developed by SYSCOM [14] can be defined.

An ATM (automatic teller machine) project developed by the Booch method is modeled using CSPL. A partial CSPL program for this project is shown in the Appendix A.

4. Experiences

Experiences gathered from the modeling and enactment, such as activity granularity and metrics are depicted in this section.

4.1. Activity granularity

The reader might feel that the processes of OMT and the Booch method shown in Figs 5 and 6 look rather simple. Let us give our experiences on this.

At first, the two object-oriented methods are modeled in some details. That is, detailed activities such as ''find classes'', ''find operations'' are separately modeled. However, it was later found that the interaction between those activities is so complex that the process program becomes unnecessarily complex. For example, in the domain analysis phase of the Booch method, constructing a class diagram requires activities such as ''find class'', ''find attribute'', ''find operation'', and ''find relationship''. During enactment, a developer looks for classes for a couple of minutes, then looks for operations for a while, and then back to look for classes. The interaction between these activities is very complex. The process program therefore becomes rather complicated. This may cause inconvenience for developers.

One approach to solve this problem would be grouping fine-grained activities to form coarse-grained activities. As shown in Fig. 7, the four activities are grouped to one coarse-grained activity as procedure *DomainAnalysis*.

Our experiences show that activity granularity is an important consideration in software process modeling. Coarse granularity reduces the degree of software process automation. On the other hand, fine-grained granularity restricts developers' creativity and productivity. Process programmers therefore need to strike the balance for proper granularity.

4.2. Metrics

A task ''Design'' of the Booch method is shown in Example 6. After its first procedure call:

1 Designer edit designDocument referring to

domainAnalysisDocument using CASETool;the design document ''designDocument'' is built.

```
task body Design is
  begin
   loop
    accept start;
     1 Designer edit designDocument referring to
               domainAnalysisDocument using CASETool;

     measure designDocument using MetricTool;

     Design_review_result :=
Design_Phase.ReviewDesign(designDocument,
                              domainAnalysisDocument);
    ......
    end loop;
   .....
   end;
```

In Example 6, the metrics statement following the procedure call is:

measure designDocument using MetricTool;

The metrics tool above automatically generates the objective metrics of the design document. Note that subjective metrics, however, are manually generated based on guidelines [9]. The objective metrics are provided to reviewers for their reference. Our experiences show that metrics from one

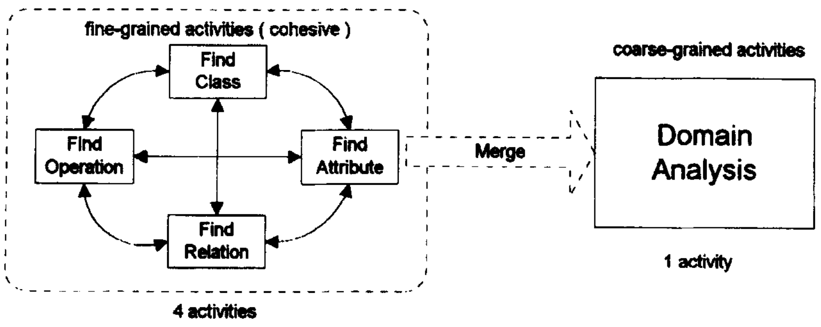


Fig. 7. Activity granularity.

project cannot convincingly determine the software quality. It is figured that only after a large quantity of metrics values is gathered from numerous projects can a ''quality criteria'' be established, such as ''attribute complexity of a class should be lower than 10'', and only with the criteria can metrics values be related to certain software quality.

5. Related work

This section reviews the expression form and task communication of some software process languages. Advantages and disadvantages of the procedural language approach used by CSPL are also discussed.

5.1. Expression form

Expression form largely determines the modeling power of a process language. CSPL uses the procedural language approach. In addition to this we have identified several other approaches: (1) Petri Net; (2) regular expression; (3) functional language; (4) rule-based language; (5) goal-directed or planning language; and (6) triggered language. These approaches are briefly described below.

The procedural language approach is used in CSPL and several other languages, such as APPL/A [15] and MDL [16]. It appears to be the paradigm that developers are most familiar with. Developers often use this kind of language such as C + + to develop a product. Therefore it is natural for the developers to use a procedural language. However, the abstraction level of this paradigm is relatively low.

The Petri Net approach is used in many process environments such as Process Weaver [6], Kernel/2r [17], and SPADE [18]. Petri Net provides a good expression form for high-level modeling, therefore it is easy to understand. Moreover, it is very suitable to model concurrent tasks, because it has a high degree of parallelism. However, Petri Net is weak in modeling low-level details. To remedy that, other constructs should be provided. For example, Process Weaver provides a co-shell to access UNIX tools and services. Moreover, Petri Net seems weak in object modeling. For example, how can the input or output of a transition be modeled in a Petri Net?

The regular expression approach is used in Hakoniwa [7]. Regular expression is easy to express rework and backtracking that are often found in software processes. It is also suitable for static analysis such as dead-lock detection. However, complex processes are difficult to specify using regular expression. Moreover, processes represented in regular expression are not very easy to read and understand.

The functional language approach is used in HFSP [19]. It applies top-down functional decomposition to obtain an activity hierarchy. The rule-based language approach used in Merlin [20] and Marvel [21] achieve a higher abstraction level than that of the procedural language approach. The latter specifies exact control flow of the processing. The former, on the other hand, specifies the form of results without revealing the lower level procedural control flow knowledge.

In a rule-based language, some rules about a process can be constructed to serve as constraints or goals for the process. This idea brings about the goal-based or planning language approach. This approach is used in Intermediate [22], [23] and Grapple [24], where mixed paradigms can be observed. In the triggered language approach used in Adele-2 [25] and EPOS [8], a database transaction may signal an event which triggers an activity. For instance, in Adele-2 and EPOS, an event can be set by a transaction to trigger an action (activity). That action may set another event which in turn triggers another action, and so on.

5.2. Task communication

A software process environment should assist developers who are working concurrently. In order to coordinate concurrent tasks, task communication is needed. From this viewpoint, a good software process language should provide constructs to support task communication. Let us see how this is done in various environments.

CSPL provides both synchronous and asynchronous communication primitives for tasks: *entry call* and *event inform*, respectively. Entry call is for synchronous communication, while event inform is for asynchronous. Synchronous communication here implies that two tasks must both reach the synchronous communication point before they can execute their activities following that point. Entry calls thus play an important role in synchronizing two tasks in a software process. In asynchronous communication, when task A sends an event to task B. Task A will not wait for task B to receive the event. Both tasks can proceed to execute their own activities.

Communication in Hakoniwa can be divided into two categories, namely *data transfer* and *task control*. Primitives for data transfer provide the mechanism to send and receive data between tasks. Primitives for task control can initiate (start) or terminate a task. All these communication primitives, however, are asynchronous.

Process Weaver provides two functions, *SendEvent* and *WaitForEvent*, to assist task communication. It, however, does not seem to provide synchronous communication primitives.

5.3. Procedural language approach

CSPL is an Ada95-like procedural process language modeling the components used in a software process, including tools, roles, objects, procedures, tasks, and so on. Since it is Ada95-like, it possesses powerful Ada95 modeling features, such as strong typing, encapsulation, inheritance, generic definition, and exception handling. These features appear useful in modeling a software process. Another important advantage of using a programming language is that the programs can be easily compiled and thus enacted. Moreover, this approach dramatically reduces the complexity of implementing an environment. Transferring the technology to the industry is thus greatly facilitated.

Compared with other approaches, the procedural language approach used by CSPL has advantages and disadvantages. This approach is qualitatively evaluated as follows.

Its advantages are:

1. Most developers are used to using procedural languages to model things. CSPL statements seem close to natural language, and therefore seem easy to use. Contrary to this, regular expression used by Hakoniwa is rather difficult to understand.
2. Procedural languages such as CSPL model a software process in a structured and object-oriented manner utilizing multiple levels of abstraction. This is rather clear compared with other approaches.
3. CSPL inherits powerful features from Ada95, such as object-type inheritance for object modeling and exception handling for dealing with unexpected situations.
4. CSPL is easy to extend. If an enhancement is needed, just add new statements such as the metrics statement mentioned earlier.

Its disadvantages are:

1. CSPL process programs are sometimes cumbersome. There are repetitive descriptions scattered around a CSPL program, such as the names in package specifications and those in package bodies.
2. CSPL lacks high-level readability due to its textual form. Graphical languages, such as Process Weaver that uses Petri Net, seem to outperform CSPL in this regard.

6. Conclusions and future work

6.1. Conclusions

In this paper we discussed the modeling and enactment of the Booch method and OMT method using the process language CSPL developed in our laboratory. The following conclusions were obtained:

1. CSPL possesses powerful modeling features. Components used in software processes such as tools, roles, activities, objects, concurrent tasks, exceptions, and so on, can be easily modeled.
2. CSPL provides rich and useful language constructs for modeling and enacting complex software processes. In our experiences the following can be easily modeled and enacted: tool binding, role management, task communication (both synchronous and asynchronous communication), and exception handling.
3. When modeling a software process, fine-grained activities can be grouped to form coarse-grained activities. This can prevent the process program from being too complicated. However, coarse-grained granularity reduces the degree of software process automation. Therefore, process programmers need to strike the balance for proper granularity.
4. Metrics can be gathered during process enactment. However, our experiences show that metrics from one project cannot convincingly determine the software quality. It is figured that only after a large amount of metrics values are gathered from numerous projects can a ''quality criteria'' be established.

6.2. Future work

Although CSPL is powerful in modeling software processes, it does have disadvantages. In our experiences, CSPL is good at low-level modeling. However, CSPL provides few supports for high-level modeling and understanding. This may result in a large process program and long development time. In addition, processes modeled by CSPL may not be as easy to understand as those modeled by Petri Net. These disadvantages may frustrate CSPL users. To remedy that, a user interface is being designed for the CSPL environment. This interface is expected to facilitate developing and understanding of CSPL process programs. It will be composed of the following components:

1. A development support component that will facilitate developing process programs. It will be designed as a window-based system that provides icons to guide the development. In addition, it will provide reusable process programs. To support that, a repository for storing reusable process programs will be designed.
2. A browsing support component that will facilitate browsing and understanding process programs. It will provide multiple windows, in which each shows a part of a process program. In addition, it will provide functions to abstract process programs. That is, a user can browse either abstract information (e.g. package specification) or detailed information (e.g. package body). With this, one can browse a process program from abstract to detailed levels and display different contents on different windows. This is expected to facilitate the understanding of process programs.

Acknowledgements

The authors wish to thank the referees for their comments and Y. L. Liu for his dedicated work. This research is sponsored by the National Science Council, Taiwan, under Grant No. NSC86-2213-E-009-025.

Appendix A Partial CSPL Process Program for ATM Project Using the Booch Method, NCTU Software Engineering Environment Lab, Programmed by Y. L. Liu

```
– Tool Definition –
tool Booch_tools is
Editor: = ''vi'';
WordProcessor: = ''interleaf'';
CASETool: = ''ROSE'';
ReviewTool: = ''reviewTool'';
MetricTool: = ''OOmetric'';
end;
– Role Definition –
role requireAnalyst is
requireAnalyst1: = ''ymchen'';
end;
role RAreviewer is
RAreviewer1: = ''jychen'';
RAreviewer2: = ''hsubj'';
RAreviewer3: = ''ymchen'';
end;
role DomainAnalyst is
DomainAnalyst1: = ''ymchen'';
DomainAnalyst2: = ''hsubj'';
end;
role DAreviewer is
DAreviewer1: = ''jychen'';
DAreviewer2: = ''hsubj'';
DAreviewer3: = ''ymchen'';
end;
role Designer is
Designer1: = ''ymchen'';
Designer2: = ''hsubj'';
end;
role DesignReviewer is
DesignReviewer1: = ''jychen'';
DesignReviewer2: = ''hsubj'';
DesignReviewer3: = ''ymchen'';
end;
– Package Specification –
package Booch_Method is

type Document is new DocType with record

reviewOption: string;
reviewResult: integer;
reviewComment: string;
authorComment: string;
metricResult: string;

end record;

– Package Specification for Problem Statement
package Problem_Statement is

...

end Problem_Statement;
– Package Specification for Requirement Analysis Phase
package RequirementAnalysisPhase is

...

end RequirementAnalysisPhase;
– Package Specification for Domain Analysis Phase
package DomainAnalysisPhase is

type DomainAnalysisDocument is new Document with record

classDiagram: NonTextType;
objectScenarioDiagram: NonTextType;
dataDictionary: TextType;
classSpecification: TextType;
workpartition: TextType;

end record;
procedure DomainAnalysis(

domainAnalysisDocument: in out DomainAnalysisDocument;
requireDocument: in RequireDocument);

function ReviewDA(

domainAnalysisDocument: in DomainAnalysisDocument;
requireDocument: in RequireDocument)
return integer;

end DomainAnalysisPhase;
– Package Specification for Design Phase –
package Design_Phase is

type DesignDocument is new Document with record

architectureDoc: TextType;
executableReleaseDoc: TextType;
classCategoryDiagram: NonTextType;
designClassDiagram: NonTextType;
designObjectDiagram: NonTextType;

end record;
function ReviewDesign(

designDocument: in DesignDocument;
domainAnalysisDocument:in DomainAnalysisDocument)
return integer;

end Design_Phase;
– Package Specification for Evolution Phase –
package EvolutionPhase
............

end EvolutionPhase;
– Package Specification for Maintenance Phase –
package MaintenancePhase

...........
```

```
end MaintenancePhase;
end Booch_Method;
– Package Body –
package body Booch_Method is
– Package Body for Problem Statement –
package body Problem_Statement is

...

end Problem_Statement;
– Package Body for Requirement Analysis Phase –
package body RequirementAnalysisPhase is

...

end RequirementAnalysisPhase;
– Package Body for Domain Analysis Phase –
package body DomainAnalysisPhase is

procedure DomainAnalysis(

domainAnalysisDocument: in out DomainAnalysisDocument;
requireDocument: in RequireDocument)

is

begin

1 DomainAnalyst edit domainAnalysisDocument referring to

requireDocument using CASETool;

end;

function ReviewDA(

domainAnalysisDocument: in DomainAnalysisDocument;
requireDocument: in RequireDocument)
return integer

is

current_time: time;
DomainAnalysisDeadLine: time;
timeout: exception;
workresult: integer;
begin

current_time: = GetCurTime;
if current_time > DomainAnalysisDeadLine

then raise timeout;

end if;
all DAreviewer review domainAnalysisDocument referring to

requireDocument using ReviewTool resulted in workresult;

return workresult;

end;

end DomainAnalysisPhase;
– Package Body for Object Design Phase –
package body Design_Phase is

function ReviewDesign(

designDocument: in DesignDocument;
domainAnalysisDocument: in DomainAnalysisDocument)
return integer

is

workresult: integer;
begin

all DesignReviewer review designDocument referring to
domainAnalysisDocument using ReviewTool resulted in workresult;
return workresult;

end;

end Design_Phase;
– Package Body for Evolution Phase –
package body EvolutionPhase is

...

end EvolutionPhase;
– Package Body for Maintenance Phase –
package body MaintenancePhase is

...

end MaintenencePhase;
end Booch_Method;
with Booch_Method;
procedure StartBoochMethod is
– Variable initialization –
– Set Time Point
DomainAnalysisDeadLine: time;
– Task Specification –
task RequireAnalysis is

entry start;

end;
task DomainAnalysis is

entry start;

end;
task Design is

entry start;

end;
task Evolve is

entry start;

end;
task Maintain is
```

```
entry start;
end;
– Task Body –
task body RequireAnalysis is
begin
...
end;
task body DomainAnalysis is
begin
loop
accept start;
loop
DomainAnalysisPhase.DomainAnalysis(
domainAnalysisDocument,requireDocument);
DA_review_result: = DomainAnalysisPhase.ReviewDA(
domainAnalysisDocument,requireDocument);
exit when DA_review_result > 0;
end loop;
if DA_review_result = 1 then
output
``*****************************************_
*****************'';
output ``******** Reviewers Decide to Advance to
Design Phase *****'';
output
``*****************************************_
*****************'';
Design.start;
end if;
if DA_review_result = 2 then
output
``*****************************************_
***************'';
output ``**** Reviewers Decide to Back to Require-
ment Analysis ***'';
output
``*****************************************_
***************'';
RequireAnalysis.start;
end if;
end loop;
– Exception Handling for Time Out –
exception
when timeout = >
output ``Time out! Starting Design Phase'';
Design.start;
end;
task body Design is
begin
loop
accept start;
1 Designer edit designDocument referring to
domainAnalysisDocument using CASETool;
measure designDocument using MetricTool;
Design_review_result: =
Design_Phase.ReviewDesign(designDocument,
domainAnalysisDocument);
if Design_review_result = 0 then
output
``*****************************************_
**********'';
output ``************** Redo Design Phase
*****************'';
output
``*****************************************_
**********'';
Design.start;
exit;
end if;
if Design_review_result = 1 then
output
``*****************************************_
**********'';
output ``***** Booch Method Software Process is fin-
ished ****'';
output
``*****************************************_
**********'';
projectFinished: = True;
exit;
end if;
if Design_review_result = 2 then
output
``*****************************************_
*****************'';
output ``***** Reviewers decide to Modify Require-
ment Analysis *****'';
output
``*****************************************_
*****************'';
RequireAnalysis.start;
end if;
if Design_review_result = 3 then
```

```
output
‘‘*****************************************_
***************’’;
output ‘‘******* Reviewers decide to Modify Domain
Analysis ******’’;
output
‘‘*****************************************_
***************’’;
DomainAnalysis.start;

end if;

end loop;

end;
task body Evolve is

begin

...

end;

task body Maintain is

begin

...

end;

– Begin of the Booch Method Software Process –
begin
– Document description setting
–Review option setting
DomainAnalysisDeadLine: = SetTime(19,7,4,96);
– Start Booch Method Software Model –
RequireAnalysis.start;
end;
```

References

[1] J.-Y. Chen, C.-M. Tu, An Ada-like software process language, Journal of System and Software 27 (1) (1994) 17–25.

[2] J.-Y. Chen, C.-M. Tu, CSPL: a process-centered environment, Information and Software Technology 36 (1) (1994) 3–10.

[3] J.-Y.J. Chen, CSPL: an Ada95-like, Unix-based process environment, IEEE Transactions on Software Engineering 23 (3) (1997) 171–184.

[4] G. Booch, Object-oriented Analysis and Design with Applications, 2nd ed., Benjamin/Cummings, New York, 1994.

[5] J. Rumbaugh, Object-oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[6] C. Fernstrom, Process Weaver: adding process support to Unix, in: Proceedings of the Second International Conference on the Software Process, IEEE Computer Society, 1993, pp. 12–26.

[7] H. Iida, K.-i. Mimura, K. Inoue, K. Torii, Hakoniwa: monitor and navigation system for cooperative development based on activity sequence model, in: Proceedings of the Second International Conference on the Software Process, IEEE Computer Society, 1993, pp. 64–74.

[8] R. Conradi et al., Design, use and implementation of SPELL, a language for software process modeling and evolution, in: Proceedings of the Second European Workshop on Software Process Technology, 1992, pp. 167–177.

[9] J.-Y. Chen, J.F. Lu, A new metric for object-oriented design, Information and Software Technology 35 (4) (1993) 232–240.

[10] R.S. Pressman, Software Engineering—A Practitioner's Approach, 3rd ed., Chap. 2, McGraw-Hill, New York, 1992.

[11] Rational, Formal Semantics of the Booch Notation for Object-oriented Analysis and Design, Rational, USA, 1994.

[12] Rational, Rose User Manual, Rational, USA, 1994.

[13] Institute of Information Industry, OOCASE User Manual, Institute of Information Industry, Taiwan, 1994.

[14] OOAid User Manual, SYSCOM Computer Engineering Co., Taiwan, 1994.

[15] S.M.Sutton Jr., , D. Heimbigner, L.J. Osterweil, APPL/A: a language for software process programming, ACM Transactions on Software Engineering and Methodology 4 (3) (1995) 221–286.

[16] J.-Y. Chen, P. Hsia, MDL (methodology definition language): a language for defining and automating software development process, Journal of Computer Languages 17 (3) (1992) 199–211.

[17] B. Holtkamp, H. Weber, Kernel/2r-A software infrastructure for building distributed applications, in: Proceedings of the Fourth International Conference on Future Trends in Distributed Computing Systems, Lisbon, 1993.

[18] S.C. Bandinelli, A. Fuggetta, C. Ghezzi, Software process model evolution in the SPADE environment, IEEE Transactions on Software Engineering 19 (12) (1993) 1128–1144.

[19] T. Katayama, A hierarchical and functional approach to software process description, in: Proceedings of the Fourth International Software Process Workshop, New York, 1989, pp. 87–92.

[20] B. Peuschel, W. Schafer, Concepts and implementation of rule-based process engine, in: Proceedings of the Fourteenth Internationl Conference on Software Engineering, 1992, pp. 262–279.

[21] G.E. Kaiser, N.S. Barghouti, Database support for knowledge-based engineering environments, IEEE Expert 3 (2) (1988) 18–32.

[22] D.E. Perry, Policy-directed coordination and cooperation, in: Proceedings of the Seventh Software Process Workshop, Yountville, CA, 1991, pp. 111–113.

[23] D.E. Perry, Enactment control in Interact/Intermediate, in: B.C. Warboys (Ed.), Proceedings of the Third European Workshop on Software Process, EWSPT 94, Villard de Lans, France (Lecture Notes in Computer Science, 772, Springer, Berlin, 1994, pp. 107–113).

[24] K.E. Huff, Probing limits to automation: towards deeper process models, in: Proceedings of the Fourth International Software Process Workshop, New York, 1988, pp. 79–81.

[25] N. Belkhatir, W.L. Melo, Supporting software development process in Adele 2, The Computer Journal 37 (2) (1994) 621–628.