A HIERARCHICAL AND FUNCTIONAL APPROACH TO SOFTWARE PROCESS DESCRIPTION

Takuya Katayama

Department of Computer Science, Tokyo Institute of Technology

1 INTRODUCTION

Research into describing software processes (such as design, development, maintenance and reuse) is attracting much attention in the software engineering community. There are a variety of views, ranging from pessimistic to optimistic, about whether it is possible to describe real and practical software processes in such a way as to guide human users in performing software activity: the process of software design, for example, is one of the most creative of human activities, and it may not be possible to achieve a complete formalisation of it at the present time. We are, however, justified in working on software process description for several reasons: every scientific study begins with description; software methods, on which a great deal of work has been done, need to be described in some language so that they can be better used and communicated; and the software industry needs some means of process description to achieve better quality control over products.

One of the key software process issues is the choice of formalisms. In general, they depend largely on the nature of the process, and there could be a variety of choices. They should, however, satisfy common requirements if they are to apply to real (not toy) software processes. They must provide a clear and understandable description, which is readable by managers and capable of evolving as a result of improvements over a long period of time by many users. They must be able to describe hierarchical design processes, as hierarchical problem decomposition is the most effective and commonly adopted strategy for designing complex processes. They must be able to describe concurrent processes, as software is usually designed by a group of designers working together, whose activities might be performed concurrently. They must also be able to represent nondeterminism or backtracking, as nontrivial design processes will include some kind of design alternatives.

The purpose of this paper is to introduce a formalism for hierarchical and functional software process description called HFSP, which satisfies the above requirements. HFSP defines software processes through hierarchical activity decomposition. Software activity is characterised only through its input and output attributes. When the relationship between input and output is not simple enough, the design activity is decomposed into subactivities together with a set of definitions of their attributes. The basic principle of HFSP, which it shares with the contractual approach taken by ISTAR [1], is to focus on the product-base characterisation of activities and their hierarchical decomposition.

0-89791-314-0/88/0005/0087 $1.50

2 A HIERARCHICAL AND FUNCTIONAL SOFTWARE PROCESS DESCRIPTION

HFSP is founded on two fundamental concepts, (1) design activity and (2) activity decomposition, explained below. Its formalism is derived from attribute grammars [2].

2.1 Activity

An activity is the unit of task in a software process. In HFSP it is assumed to be completely characterised by its input and output attributes, which denote, for example, requirement specification, design document, analysis report, program code, or any other software product related to the activity and stored in the object base. The activity may be simple, performed by a designer using some tool, or it may be a complex task which has to be decomposed into subtasks. Activity A with input attributes $x_1,\ldots,x_n$ and output attributes $y_1,\ldots,y_m$ is denoted by

$$A\ (\ x_1,\ldots,x_n \mid y_1,\ldots,y_m\ )$$

where $x_1,\ldots,x_n$ and $y_1,\ldots,y_m$ denote objects in the object base. Execution of A is performed functionally and it does not refer to or change any global object. That is, the content of $y_1,\ldots,y_m$ after the execution of A does not depend on any object other than $x_1,\ldots,x_n$ and it does not change the content of any object other than $y_1,\ldots,y_m$.

2.2 Activity Decomposition

If an activity is simple enough to be performed by invoking tools, its execution is left to the human activity of doing the job by using the tools. If it is not, however, the activity must be decomposed into sub-activities. We proceed with this activity decomposition until every activity becomes a primitive one performed by some tool, or performed by human mental activity such as thinking, planning or decision making.

Activity decomposition must specify how an activity is decomposed into other activities and what relationship holds among attributes of the activities involved. Suppose an activity A is decomposed into sub-activities $A_1,\ldots,A_k$. We have to associate a set E of attribute definitions and decomposition condition C with this activity decomposition, and it is denoted as follows.

$$A \Rightarrow A_1,\ldots,A_k \text{ when } C \text{ where } E.$$

The set E of attribute definitions specifies which objects are the inputs of subactivities and how to get the result of the main activity A when the subactivities A_js come up with their execution results. That is, E contains the definitions for input attributes of sub-activities A_j $(j = 1,\ldots,k)$ and

output attributes of the main activity A. Every attribute definition is of the form

$$a = f(a_1, a_2, \ldots)$$

where a is the attribute to be defined, a_1, a_2, ... are other attributes in the decomposition, and f is a function which is usually executed by invoking tools.

As an activity might be decomposed in different ways depending on the values of its input attribute objects, it is necessary to specify when this decomposition can take place. It is expressed by the decomposition condition C, which is a predicate of attributes of A, A_1, ..., A_k.

2.3 Concurrency

Concurrency is essential for describing design processes for big software, as such software design is usually performed by a group of designers working together. Their activities have to be synchronised, and they have to access the common object base. In HFSP, concurrency is expressed through attribute dependency.

Consider the activity decomposition

$$A \Rightarrow A_1, \ldots, A_k \quad \text{when C} \quad \text{where E.}$$

When A is decomposed into sub-activities A_1, ..., A_k, it does not mean that they have to be executed in that order. Rather, they are allowed to be executed as concurrently as possible. In general, an activity can be executed as soon as its input attributes become available. Thus two activities A and B can be executed concurrently if there is no data dependency among their attributes. In contrast, if there is dependency among them, they have to be executed in the order determined by the dependency.

2.4 Non-determinacy

Nondeterminacy is useful in describing design alternatives. HFSP expresses it through decomposition conditions. Consider the activity decomposition

$$A \Rightarrow A_1, \ldots, A_k \quad \text{when C} \quad \text{where E.}$$

The decomposition condition $C = C(a_1, \ldots, a_h)$ is usually specified in terms of input attributes $a_1, \ldots, a_h$ of the main activity A. This means that C can be

evaluated before decomposition takes place, and decomposition proceeds deterministically. However, if C is specified using some output attributes of A or A_i, C cannot be evaluated until A or A_i produces the output attribute objects, and we cannot know the correctness of applying the decomposition until it has been actually applied. If C turns out to be false, we have to choose another decomposition whose C might be true.

Though HFSP can represent backtracking and nondeterminism elegantly as shown above, they should be used cautiously as they will introduce a heavy revision control problem in the object base.

2.5 Activity execution

Given a set D of activity decompositions, an initial design activity A_0, and its input attribute objects V, execution of A_0 with V starts with finding an activity decomposition in D whose main activity coincides with A_0 and whose decomposition condition C is true for V. If such a decomposition is found, A_0 is decomposed into subactivities A_is and we repeat this process for each A_i. If A_i is simple enough and does not need to be decomposed, it is executed by invoking tools associated with it. Execution of A_0 ends when all the primitive activities derived from A_0 have been finished by using tools, and the output attributes of A_0 have been obtained. The entire process of executing A_0 can be seen as growing a tree representing the applied activity decompositions and evaluating attribute values on the decomposition tree.

3 ACTIVITY EXECUTION MANAGEMENT

The biggest difference between usual program descriptions and software process descriptions is, of course, that primitive operations in programs are usually tiny operations and are executed automatically in the CPU, while software process operations are big operations executed by human users invoking software tools. This difference forces us to add one more aspect to HFSP. This is activity execution management for the scheduling, execution and monitoring of activities: it undertakes the following.

(1) Analyse dependencies among activities from activity decomposition descriptions, to identify activities that could or must be executed in serial or parallel.

(2) Initiate execution of activities. If they are primitive ones, and their input attributes are all available, ask the user to execute them using given tools. If they are not primitive and need to be decomposed, select possible decompositions.

(3) Handle concurrency control if there are several activities which can be executed concurrently.

(4) Handle backtracking control if there are several decompositions that could be applied to a current activity. That is, after selecting a decomposition, if the selection is found to be wrong, pick another alternative. The trace of the wrong execution should not be discarded, but should be recorded as a failed branch in the process decomposition tree; as such it can be used for later process analysis.

(5) Keep track of the entire activity execution and show users the current process status. This could be done by displaying the decomposition tree, augmented for example by information on the status of activities, failed decomposition choices and other managerial information.

(6) Interface with the object management system. When an activity is initiated, it retrieves its input attributes from the object base, and when execution is complete it stores output attributes in the object base. Objects to be stored are specified in the activity decomposition

$$A \Rightarrow A_1, \ldots, A_k \text{ when } C \text{ where } E \text{ object } O.$$

O is a list $a_1, \ldots, a_n$ of output attributes of A, A_i which are to be stored. These objects are stored, with their execution paths in the decomposition tree as part of their identification.

Compared to conventional programming language interpreters, activity management requires more flexible control during the execution of activities. AI language researchers have recently introduced the concept of **reflection**, which allows programs to refer to and change their execution state, and which could provide flexible execution modes. This concept of reflection might be useful in process description.

4 THE HFSP-BASED SOFTWARE DESIGN ENVIRONMENT AND ITS PROTOTYPING

An HFSP-based software process environment is being constructed using the functional language AG and its environment SAGE, developed at the Tokyo Institute of Technology [3]. AG is designed to write functional programs for hierarchical and structure-oriented problems, and is based on concepts of modules and their decomposition. A module represents a function, and AG performs computation by repeated module decomposition [4].

As typical software processes, JSP and JSD have been described in AG. Activity decomposition descriptions are handled by AG's module decomposition capability with only slight modifications, though we needed additional modules for window control, concurrency control and tool invocation (which are left to the activity execution management system in HFSP). As we have not yet constructed an object base, the Unix file system was used.

The purpose of the prototype is, in the first place, to make sure that HFSP is suitable for process description and, in the second place, to try to prototype the conceptual model of the entire software process environment including its products, process and tool collection model [5], [6]. Though the prototyping effort is not complete, we consider that the hierarchical and functional approach could provide a good formalism for software process description.

REFERENCES

[1] **Dowson M.** "ISTAR - an integrated project support environment". Proceedings of the Software Engineering Symposium on Practical Software Development Environments. 1987.

[2] **Knuth D E.** "Semantics of context-free languages". Mathematical systems theory 2 (2), 1968, pp 127-145.

[3] **Shinoda Y and Katayama T.** "Attribute grammar based programming and its environment". Proceedings of the 21st Annual Hawaii International Conference on System Sciences, vol II, software track, 1988, pp 612-620.

[4] **Katayama T.** "HFP: hierarchical and functional programming". Proceedings of the 5th International Conference on Software Engineering, 1981, pp 343-353.

[5] **Riddle W E.** "Software Designer's Associates: a preliminary description". Proceedings of the 20th Annual Hawaii International Conference on System Sciences, 1987, pp 371-381.

[6] **Kishida K, Katayama T, Matsuo M, Miyamoto I, Ochimitzu K, Saito N, Sayler J H, Torii K, Williams L G.** "SDA: a novel approach to software environment design and construction". Proceedings of the 10th International Conference on Software Engineering, 1988.