Computers in Human Behavior, Vol. 10, pp. 139–154, 1994
Printed in the U.S.A.

0747-5632/94 $6.00 + .00

Plan-Based Delivery Composition in Intelligent Tutoring Systems for Introductory Computer Programming

Hein P. M. Krammer, Jeroen J. G. van Merriënboer, and Rudolf M. Maaswinkel

University of Twente

Abstract — *In a shell system for the generation of intelligent tutoring systems, the instructional model that one applies should be variable independent of the content of instruction. In this article, a taxonomy of content elements is presented in order to define a relatively content-independent instructional planner for introductory programming ITS's; the taxonomy is based on the concepts of programming goals and programming plans. Deliveries may be composed by the instantiation of delivery templates with the content elements. Examples from two different instructional models illustrate the flexibility of this approach. All content in the examples is taken from a course in COMAL-80 turtle graphics.*

INTRODUCTION

One of the main features of an intelligent tutoring system (ITS) is its capability of explicit modelling of instruction (Burns & Caps, 1988; Park, Perez, & Seidel, 1987), which may open the way to vary instruction independently from its content (O'Neil, Slawson, & Baker, 1991). This is also the main goal of the project Intelligent Tutoring Shell System for Executable Languages (ITSSEL), which is conducted at the University of Twente (Krammer, 1990; Maaswinkel & Offereins, 1990). The project aims at the development of a shell system for the generation of ITSs that may vary with regard to both their content and their applied instructional model.

An instructional model contains the pedagogical knowledge that is used to make decisions about instructional methods, sequencing of content and deliveries, types of delivery, degree of learner control, and so on. Part of the instructional model is

Requests for reprints should be addressed to Hein P. M. Krammer, Department of Instructional Technology, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.

the so-called *instructional planner*, which contains knowledge of how to plan and sequence the instruction (Halff, 1988; MacMillan, Emme, & Berkowitz, 1988; Murray, 1989). The instructional planner should support sequential decisions regarding both the content of instruction and the deliveries of instruction.

Three instructional methods can be distinguished — namely, organizational strategies, delivery strategies, and management strategies. According to Reigeluth (1983), the delivery strategy variables are elemental methods for conveying the instruction to the learner and/or for receiving and responding to input from the learner. Analogous to this distinction we will use the concept of a *delivery*. Examples of deliveries as conceived here are examples, analogies, exercises, hints, questions, explanations, tasks, and tests.

The part of the instructional planner which is concerned with the sequencing and composition of the deliveries is called the *delivery planner* (Brecht, MacCalla, Greer, & Jones, 1989; Wasson, in press). The composition of deliveries is based on "delivery templates" that may be instantiated with content elements that are stored in the ITS. Figure 1 illustrates the relationships between the parts of the ITS that are involved in the composition of deliveries.

The design of an instructional planner is a major problem if the ITS must contain a domain-independent instructional model, since content elements and delivery templates are addressed at the same time. In the ITSSEL project, three main approaches are used to tackle this problem. First, a taxonomy of content elements has been developed by which the content can be specified without referring to the content itself; the content elements may be used to plan the content and to compose the deliveries. Another example of this approach may be found in Merrill's Component Display Theory (CDT; Merrill, 1983) and more recently in his instructional transaction theory (Merrill, in press). Second, the ITSSEL project allows for

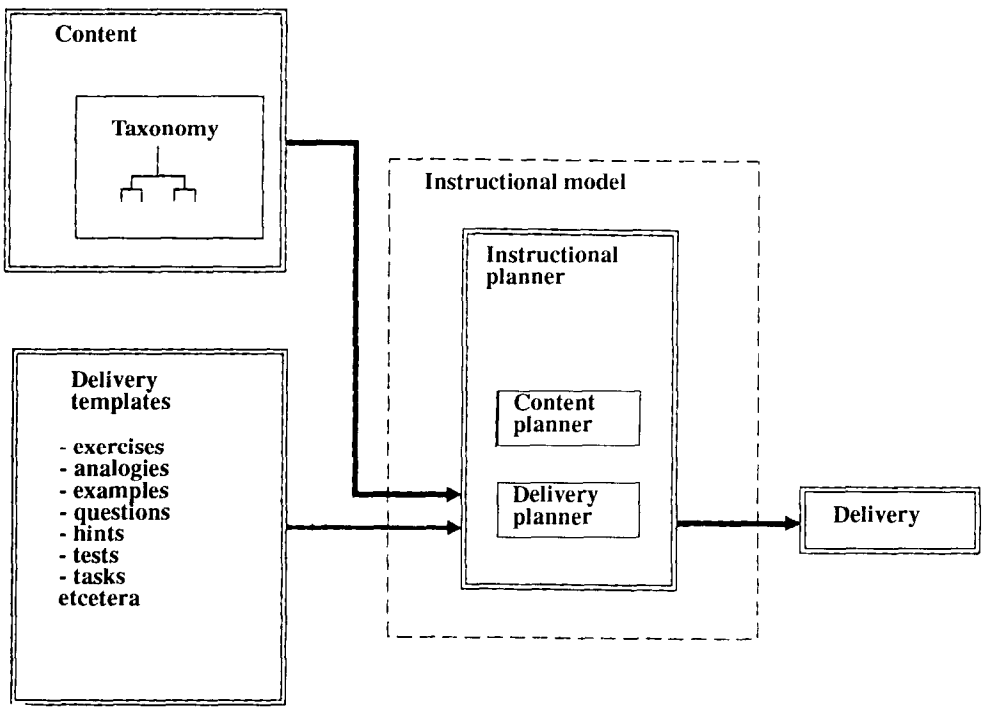

Figure 1. Relationships between ITS elements involved in delivery composition.

the architecture of the system to implement separate modules for content planning and delivery planning. And finally, the variability of domains to be covered by the ITS shell is restricted to executable languages; as a result, the domains that are covered are similar enough to be treated under the same instructional models.

In this article, the taxonomy of content elements will be presented. The taxonomy is based on the ideas of programming goals and programming plans as propagated by Soloway and his coworkers (Johnson & Soloway, 1985, 1987; Soloway & Ehrlich, 1984; Spohrer, Soloway, & Pope, 1985), and extends the possibilities which have been introduced in the ITS Bridge (Bonar & Cunningham, 1988a). The taxonomy provides for a complete description of all external actions of the system and may be used as a basis for both the formulation of the content planning rules (Krammer & Dijkstra, 1992) and the delivery planning rules (Van Merriënboer, Krammer, & Maaswinkel, 1992). After the presentation of the taxonomy, the applicability to compose a variety of deliveries, under two different instructional models, will be illustrated with examples from an introductory programming course for turtle graphics in COMAL-80 (Christensen, 1982). As the concepts of programming goals and programming plans have been worked out for different programming languages (e. g., Gegg-Harrison, 1991; Johnson & Soloway, 1987; Rist, 1989; Wiedenbeck, 1986), the ideas of this paper can be transferred to a variety of languages and subjects.

A TAXONOMY OF CONTENT ELEMENTS FOR INTRODUCTORY COMPUTER PROGRAMMING

Figure 2 presents a taxonomy of content elements in an introductory programming course and an overview of their relationships. The content elements may be (a) zero-order, intermediate, or final problem solving products (viz., problem text, goal decomposition, plan specification, or code) or (b) learning elements. The learning elements can be distinguished into (a) subject matter (viz., programming goals, programming plans, and syntax rules) and (b) strategies (viz., analysis heuristics, plan principles, discourse rules, and test heuristics). All content elements are related to the main steps in the problem solving process (analysis, design, implementation, and testing).

Subject Matter

According to our taxonomy, the subject matter consists of syntax rules, programming plans, and programming goals. The syntax rules and the language commands, which are treated as belonging to the syntax rules, will not be extensively discussed here. Some examples of syntax rules are: To every FOR belongs an ENDFOR; a program should end with END; text after // on the same line does not belong to the program, and so forth.

A programming plan is a schematic description of the structure of a particular piece of code which reaches a specific goal of the program. An example is the plan to count how many times a loop has been passed. It consists of two parts: an initialization part before the loop, and an update part within the loop. In the initialization part a counting variable is set to zero (*counter := 0), and in the update part this same variable is increased by one (*counter := *counter + 1). A programming plan may contain parameters and free variables (usually labeled by asterisks, in this case

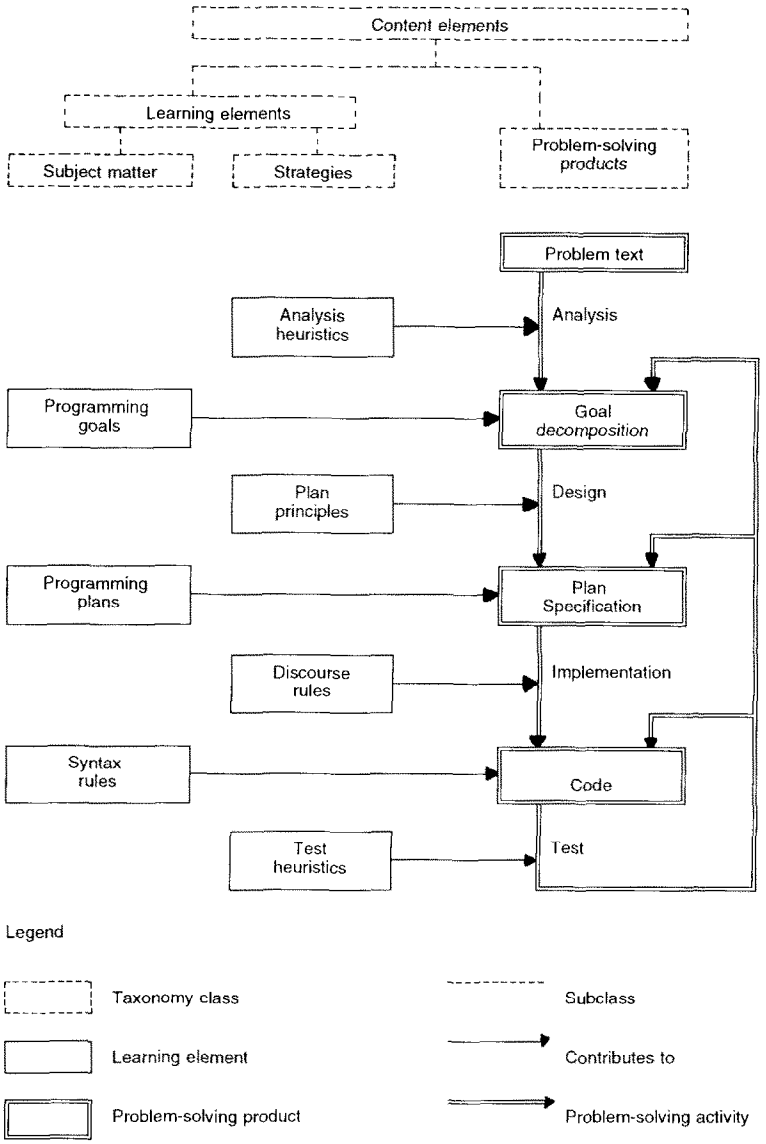

Figure 2. Taxonomy of content elements and overview of relations to problem solving activities.

*counter), and may refer to other plans or may itself contain labels (labeled by #, in this case #initialize: and #update:) to which other plans can refer, and contains itself a reference to another plan which contains a loop.

A programming goal is an element of the semantics of the programming language and specifies potential computer behavior which can be specified in a program. Usually, a goal can be reached by more than one programming plan. For instance, the goal to draw a square can be reached by several plans with different contents of the FOR loop — namely, (a) FORWARD (*length); LEFT (90); (b) BACK (*length); RIGHT (90); (c) LEFT (90); FORWARD (*length), and so on. Students must learn how to select the goals from the given problem text. In most cases, special terms in the problem text will cue the students to particular programming goals.

Problem Solving Products

Products of the problem solving process are problem texts, goal decompositions, plan specifications, and program code. In our taxonomy, the problem text itself is interpreted as a zero-order product. A goal decomposition is just a list of the goals that must be reached by the program.

A plan specification is a formal description of the plans that are used in the algorithm as well as the interrelations that occur between those plans. This specification is much more exact than a goal decomposition: Exactly one plan is selected to reach each goal, the relative positions of the plans in the program are indicated, and the free variables and the parameters of the plans are specified. It is possible to derive the program code from the plan specification in a straightforward way.

In Figure 3, an example problem is presented along with its goal decomposition, plan specification, and program code. It should be noted that the form in which the elements of the plan specification are presented here is not identical to the way they are represented in the system, nor identical to the way they are presented to the student. The plan specification is built from plan names (e.g., SetConstants), free variables (e.g., *incr), labels (e.g., #process, #exit), references to other plans (e.g., after 1), or references to parts of other plans (e.g., in 2[#process], before 2[#exit]).

Problem Solving Strategies

According to our taxonomy, strategies consist of analysis heuristics, plan principles, discourse rules, and test heuristics. These strategies are assumed to steer the student's behavior while solving the problem. The strategies are ordered according to the main steps in the problem solving process — namely, (a) analyzing the problem, (b) designing an algorithmic solution, (c) implementing the solution, and (d) testing, debugging, and maintaining the program.

Analysis heuristics are rules of thumb for the selection of programming goals which have to be reached by the program. These strategies help the students to recognize the goals that must be extracted from the problem text. Some strategies are of a rather general nature. An example of a general strategy is: "If the problem description does not contain a picture of what the final product of the program should look like, then try to draw a picture yourself with paper and pencil." However, in most cases the rules of thumb will be specific for a particular goal, as in the following example: "If it is required that the program should be used in a generic way under different circumstances, than use the goal 'Set Constants.'"

Plan principles are either specific rules or rules of thumb that prescribe how to derive a plan specification from a goal decomposition. The principles pertain to the set of the programming plans that may be used to reach particular goals, and the possible parameters that belong to each of those plans. Furthermore, the principles contain rules of thumb that may be used to properly link up the plans; they suggest how to find the parameters of the plans and how plans should refer to one another. Some of the rules are of a general nature, such as "Start and finish each plan with the turtle in an upright position, eventually recombining turns to get efficient code." An example of a specific plan principle is: "Combine in a count-how-many plan initialization to 0 with update before the process part of the loop, and initialization to 1 with update after the process part of the loop."

Discourse rules (Joni & Soloway, 1986) are rules of thumb which prescribe how to derive programming code from a plan specification. These rules mainly have to do with the readability and comprehensibility of the program. Some aspects of interest are the proper choice of names for variables and procedures, the structuring

Problem text

Write a program to draw a picture consisting of radii from the centre. The 1st radius is drawn upright and has a length of 100 units. Each following radius makes an angle of, say, 7 grades to the right with and is, say, 0.5 units shorter than its predecessor. Drawing stops when next radius to be drawn makes an angle of less than 1 grade with first radius, or the next radius has negative length. Finally, the number of radii is printed on screen.

Plan specification

1. SetConstants (*incr := 7, *decr:= 0.5)
2. Loop (#process, #exit): after 1
3. DrawRadius (*radius, *incr, right): in 2(#process)
4. RunningTotal (*angle, *incr): in 2(#process), after 3, before 2(#exit)
5. RunningRest (*radius, *decr): in 2(#process), after 3, before 2(#exit)
6. CountHowMany (*counter): in 2(#process), after 3, before 2(#exit)
7. Condition (*angle>360-*incr, *angle:=*angle-360): in 2(#process), after 6, before 2(#exit)
8. Output (*counter): after 2

Goal decomposition

set constants
loop
draw
draw {same radius backward}
turn {7 grades}
running total {angle}
running rest {radius}
count {number of radii}
condition {if round, delete 360 from angle}
output {number}

Code

```
001 // Spiral of radii
002 // Draws radii from cen
003 // tre with fixed angle
004 // between. First radius
005 // has length 100 and is
006 // upright. All next ra
007 // dii are shorter by a
008 // fixed amount. When
009 // close to 1st radius,
010 // stops & prints number
011 a'incr:=7
012 r'decr:=0.5
013 radius:=100
014 USE TURTLE
015 HIDETURTLE
016 angle:=0
017 counter:=0
018 LOOP
019   draw'radius
020   radius:=radius-r'decr
021   angle:=angle+a'incr
022   counter:=counter+1
023   IF angle>360-a'incr THEN angle
        :=angle-360
024   EXIT WHEN ABS(angle)<1
025 ENDLOOP
026 PRINT counter
027 END
028 PROC draw'radius
029   FORWARD(radius)
030   BACK(radius)
031   RIGHT(a'incr)
032 ENDPROC draw'radius
```

Figure 3. Example problem text with goal decomposition, plan specification, and a solution in code.

of the program by means of procedures and functions, the use of indentation, and the addition of comments.

Finally, test heuristics are rules of thumb that suggest how to test the correctness of a program (e.g., "If the program expects a string of data as its input, for instance a name, then test the behavior of the program by entering a string of zero length").

EXAMPLES OF PLAN-BASED DELIVERY COMPOSITION

A variety of deliveries may be defined in terms of the content elements of the presented taxonomy. If an ITS contains the content elements in its knowledge bases it is possible to have the system construct the deliveries automatically. In fact, two

content elements need not to be stored in the system's knowledge bases: The goal decomposition and the program code can be derived by the system itself if it has the plan specification available.

The potential deliveries and the exact format of the deliveries depend on several factors which are determined by the instructional model, or the instructional theory on which this model is built. To illustrate the possibilities, examples of delivery composition for two instructional models will be presented in the next sections. The first case is based on a simplified version of Merrill's Component Display Theory (CDT; 1983). The second case has been taken from the Completion Strategy as promoted by Van Merriënboer (1990b).

Delivery Composition Based on a Subset of Merrill's Component Display Theory

In CDT, three delivery attributes are relevant — namely, the initiative, the communication form, and the instructional mode. The *initiative* to the delivery may come from (a) the student or (b) the system. The *communication form* indicates if the delivery is in (a) "inquisitory" form (i.e., some kind of question) or in (b) "expository" form (i.e., an assertion). The *instructional mode* may be (a) a generality or (b) an instance of this generality. Whereas the generality is the formal content of the learning element, an instance illustrates the content of the learning element. Together, the communication form and the instructional mode determine the so-called "primary presentation forms." The other keystone of Merrill's CDT theory (i.e., its two-dimensional taxonomy of content and performance) will not be considered here; instead, all illustrations will be related to our taxonomy of content elements as presented above.

Delivery templates. According to CDT, each interaction between student and system is formed by one of the eight delivery templates that are generated by the three dimensions. For instance, the student may ask for an illustrative example (initiative: student; communication form: inquisitory; instructional mode: instance). Likewise, the system may present a rule (initiative: system; communication form: expository; instructional mode: generality). At other moments the system will ask to give an example (initiative: system; communication form: inquisitory; instructional mode: instance), followed by a student answer (initiative: student; communication form: expository; instructional mode: instance).

Instantiation of delivery templates. Usually, the system-initiated expository generalities can be simply composed from the available text stored in the ITS. For instance, the generality for a programming plan may be instantiated by (a) the name of the plan, (b) the programming goal that is reached by the plan, (c) the scheme of the plan, and (d) the labels and parameters used in the plan. Or, the generality for a programming goal may contain (a) the name of the goal, (b) a description of the intended computer behavior, (c) the programming plans reaching the goal, and (d) cues in the problem text which usually are associated with the goal. It is supposed that all this information is prestored (literally or in coded form) in the ITSs knowledge bases.

However, some deliveries must differ in format from the stored text; a "didactical specification" may be necessary to reach a format that is easily understood by the students. Two examples may illustrate this general proposition. First, a goal decomposition may be delivered in a natural language–like format, whereas it is stored in coded form within the system. By matching each goal in the knowledge

base to a unique natural language phrase the system may construct a delivery of a goal decomposition similar to the first phase of the Bridge system (Bonar & Cunningham, 1988b). Second, the delivery of a plan specification may also profoundly differ from the way plans are presented so far. Formal plan specifications like the ones used in Figure 3 are clearly useless for students. Perhaps a graphic representation of plans is best suited for students. Again, the ITS Bridge illustrates what could be done, even with two different examples of graphic representations (Bonar & Cunningham, 1988a, 1988b).

An example of a system-initiated expository instance is presented in Figure 4, in which the discourse rule on structuring ("Combine actions which logically belong together into procedures") is illustrated for the problem as presented in Figure 3. The delivery is composed from a template which serves for all system-initiated expository instances of discourse rules. The delivery template for an instance of a discourse rule will consist of (a) the concerning part of the plan specification for a problem, (b) the matching programming code in which the concerning part is marked or highlighted, and (c) connecting explanatory text which is applicable for all instances of this particular analysis heuristic.

The delivery of a system-initiated expository instance for a programming plan will consist of the name of the plan as well as a program in which this plan is applied. The text concerning the plan is marked or highlighted in the program. An explanation may be added which can take essentially the same form for all programming plans.

System-initiated inquisitory deliveries offer students the possibility to answer questions in the form of student-initiated expository deliveries. So, these two types of delivery are coupled. In this case the student has available in the editor the possibility to mark or highlight words or phrases. Using this possibility (which is in effect the student-initiated expository delivery) the student may present his answer.

It is expected that all student-initiated expository deliveries are built-in answers to system-initiated inquisitory deliveries as described above. On the other hand, the student-initiated inquisitory generalities and instances need a special provision. As

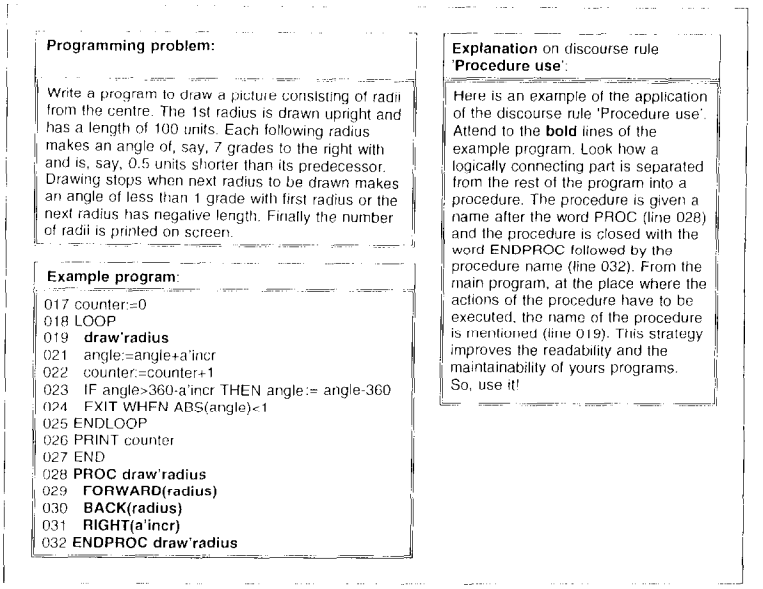

Figure 4. Example of an instance delivery for a discourse rule.

all deliveries are based on a limited set of content elements, these deliveries need no natural language interface and can simply be constructed in the form of menus.

Delivery Composition Based on the Completion Strategy

As a second example of our approach, the delivery composition will be illustrated for an instructional model that is usually referred to as the Completion Strategy (Van Merriënboer & Krammer, 1987, 1989; Van Merriënboer & Paas, 1990). In contrast to most traditional training strategies in which students independently have to design and code increasingly complex computer programs, the Completion Strategy requires the students to complete or extend increasingly larger parts of computer programs which are incomplete but otherwise well designed and easily readable. In several experiments (Van Merriënboer, 1990a, 1990b, in press; Van Merriënboer & De Croock, in press), the Completion Strategy yielded higher learning outcomes than more traditional strategies that were focusing on the students' unconstrained generation of new computer programs. In the following sections, a brief description will be given of the used delivery templates, the content elements that serve to instantiate those templates, and an automated system to sequence and construct deliveries according to the Completion Strategy.

Delivery templates. On a sufficiently high level of abstraction, only one type of delivery is used in the Completion Strategy — namely, the *completion assignment.* Completion assignments should be seen as the basic building blocks of instruction. They always consist of a description of a programming problem; in addition, they may contain one or more of the following delivery templates: (a) examples, (b) explanations, (c) questions, and (d) instructional tasks. Examples may either provide a complete solution to the posed programming problem in the form of a well-designed, easily readable computer program, provide an incomplete solution in the form of a partial computer program, or be absent. Explanations refer to new features of the programming language or programming task; in the Completion Strategy, these new features are always illustrated by (parts of) the (in)complete example program. In addition, questions may be asked on the working and the structure of features of the (in)complete example program. Finally, instructional tasks refer to the exercises that are given to the student. Possible tasks are to solve the posed programming problem (if no example is provided), to complete an incomplete example, or to extend or change a complete example. In the following section, we will elaborate on the instantiation of those delivery templates with content elements in order to plan and compose the deliveries (i.e., completion assignments).

Content elements. The Completion Strategy uses, in its present form, 5 of the 11 content elements that were discussed in relation to our taxonomy of content. First, each completion assignment always contains a *problem text* (i.e., a description of the programming problem in natural language). It should be noted that the problem text describes the programming problem that must be solved by the computer program; designing and coding this program is not necessarily the task that must be performed by the student. Thus, a clear distinction is made between the problem text and the instructional task(s) the student has to perform.

Second, completion assignments may provide the student with a product in the form of *program code,* which is used to instantiate an example. Simply stated, three possibilities emerge. First, the example may be instantiated with program code that represents a complete solution to the programming problem as described in the problem text; then, the student is confronted with a fully worked-out, well-

designed, and easily readable computer program that offers a model solution to the given programming problem. Second, the example may be instantiated with program code that only offers a partial solution to the programming problem; in this case, the example takes the form of an incomplete computer program. Finally, the example may be instantiated with no program code at all; in this case, no example is presented to the student.

Third, completion assignments may provide the student with several learning elements. In the subject matter category, these pertain to *programming plans* and *syntax rules*; in the strategies category, these only pertain to *discourse rules*. The learning elements are used for the instantiation of both explanations and questions. As a requirement of the Completion Strategy, explanations or questions on programming plans, syntax rules, or discourse rules are always coupled to elements in the example (i.e., either an incomplete or a complete program) that illustrate the application of these plans or rules. Thus, the "generalities" only occur in combination with their "instances" (cf., Merrill, 1983). As a consequence, the instantiation of the delivery templates requires two actions: The delivery template must be coupled to a content element (yielding, e.g., an explanation on a particular plan or a question on a particular discourse rule), and the content element must be coupled to its instance as applied in the example program.

Given this constraint, explanations on programming plans present a description of a particular plan to students (e.g., its name, related programming goal, scheme or graphic representation, labels, parameters, etc.) and illustrate this plan by marking or highlighting its instance in the example. In the same way, explanations on syntax rules (including new language commands) present new syntactical information to students and illustrate this information in the example; explanations on discourse rules present information that pertains to good programming practice and, again, illustrate this information in the example (cf., Figure 4).

As mentioned before, the same content elements may also be used to instantiate questions. With regard to "inquisitory instances," one may present the student with (the name of) a particular programming plan, syntax rule, or discourse rule and ask him or her to identify this plan or rule in the example by marking it. With regard to "inquisitory generalities," one may mark particular elements in the example and ask the student to identify or describe the plan or rule that has been applied. Obviously, it is possible to generate a very broad range of deliveries within the Completion Strategy. Figure 5 provides an overview of the elements that may be presented in completion assignments. Conventional programming problems and conventional worked examples should simply be seen as extreme cases: A conventional programming problem consists of a problem text and the task to solve the posed programming problem (example program, explanations, and questions are not presented). A conventional worked example consists of a problem text and an example that provides a complete solution to this problem (explanations, questions, and tasks are not presented). In between, innumerable combinations are possible. For instance, a particular completion assignment may contain a problem text, an example that provides an incomplete solution to this problem, two explanations on a new plan and a new discourse rule that are illustrated in the incomplete program code, a question on a particular syntax rule that is used in the incomplete program code, and the task to complete the incomplete example.

Composing completion assignments. Van Merriënboer, Krammer, and Maaswinkel described CASCO (Completion Assignment Constructor; for a complete description, see Van Merriënboer, Krammer, & Maaswinkel, 1992), which is an automated

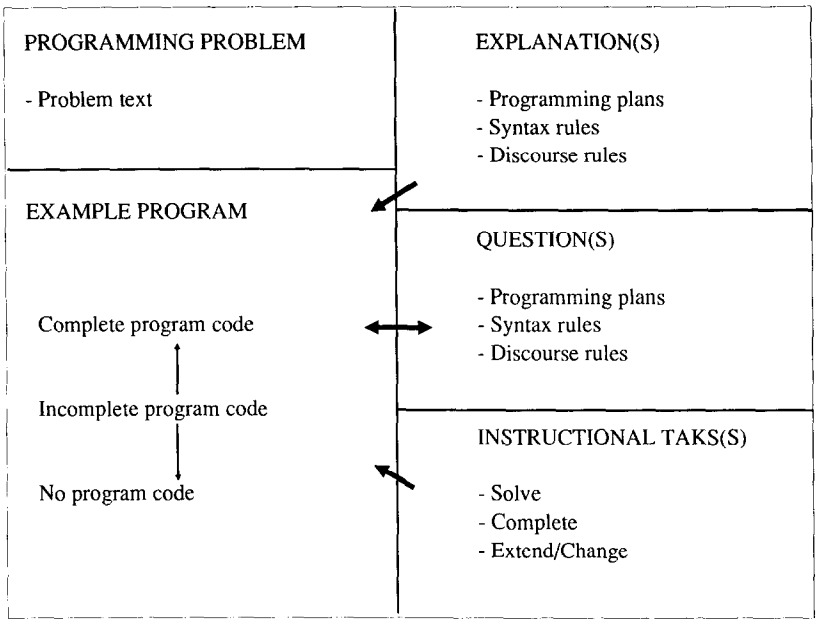

Figure 5. Elements of a completion assignment.

system for the sequencing and construction of completion assignments. Essentially, it may be viewed as a system that instantiates the distinguished delivery templates (examples, explanations, questions, and tasks) in order to compose completion assignments that are tailored to the individual needs of a particular student. In order, we briefly describe its knowledge bases that contain the elements necessary to compose the next delivery, its content planning module, and its delivery planning module.

Two knowledge bases may be distinguished: (a) a *domain model* with an overlay student profile and (b) a problem database. The domain model (i.e., the model of the knowledge that must be taught) consists of a comprehensive library of programming plans on which part-of/prerequisite relations, conflict relations, and pedagogical priority relations are defined. The student profile is a simple partitioning of this domain model in four sets. Set 1 is formed by the plans that have not yet been presented to the student; Set 2 is formed by the plans that have been presented to the student, but which the student has not yet applied; Set 3 is formed by the plans that the student has exercised a few times, but that are not yet fully learned, and Set 4 is formed by the plans that have been correctly applied for many times and are assumed to be automated. The *problem database* consists of a large but finite number of programming problems which can be presented to a student. For each problem, the following information is available in the database: (a) a problem text; (b) a complete solution for this problem in the form of a plan specification, from which the goal decomposition and program code can be derived; (c) prestored explanations on the plans, syntax rules, and discourse rules that are used in the program; (d) prestored questions on these plans, syntax rules, and discourse rules, and (e) a set of possible instructional tasks that may be presented to the student.

CASCO's *content planning* module (cf., Barr, Beard, & Atkinson, 1976) generates three sets of candidate plans: (a) a set of candidate plans to be presented next, (b) a set of candidate plans to be exercised in a "narrow" context, and (c) a set of

candidate plans to be further exercised in a wide variety of situations. Each of the sets is generated on the basis of the relations that are defined on the plans in the domain model. For example, a plan may only be added to the set of candidate plans to be exercised in a narrow context if all plans that have a conflict relation with this plan are either not yet presented (i.e., belong to Set 1) or have been applied correctly for many times (i.e., belong to Set 4). In order to select the most suitable problems from the problem database, all problems that require the presentation or use of too many new plans are excluded. From the remaining problems, the problem is chosen that optimizes the "deliverance possibility," indicating if the problem contains many plans that open the way to subsequently use many other problems, and the "pedagogical priority", indicating the desirability to teach particular plans early in the learning process.

CASCO's *delivery planning* module subsequently composes a completion assignment for the selected problem. The composition of the completion assignment depends on the candidate sets as well as the set of plans that is necessary to solve the selected problem. The instantiation of the delivery templates is governed by four rule sets: (a) example rules, (b) explanation rules, (c) question rules, and (d) task rules. The example rules are particularly important to the Completion Strategy and serve to instantiate an example with program code; in effect, they delete portions from the solution for the programming problem as specified in the problem database. For instance, one of the rules determines that *if* none of the plans that are necessary to solve the posed problem are in one of the three candidate sets (i.e., they are all assumed to be automated), *then* all plans are deleted from the solution; thus, the student will be confronted with a conventional programming problem for which no partial solution is provided. The following rule may serve as a second example: *If* exactly one plan of the selected problem is in the candidate set of plans to be exercised in a narrow context, and no more than two plans are in the set of candidate plans to be presented for the first time, *then* delete this one plan from the solution.

The explanation rules and question rules serve to instantiate either explanations or questions with content elements that pertain to programming plans, syntax rules, or discourse rules. An example of a simple rule that instantiates an explanation on a new plan is the following: *If* one or more plans of the presented example are in the candidate set of plans to be presented for the first time, *then* provide the (prestored) explanation on the working of the plan(s) and highlight the realization of this plan in the example that is provided to the student. Finally, the task rules specify the instructional task that will be presented to the student (either generating a complete program, completing a program, or changing or extending a complete program).

Figures 6 and 7 provide two examples of completion assignments that may be composed by the system. It concerns the same problem, for which completion assignments are composed early in the learning process when much has to be explained to the student, and later in the learning process when the student is able to complete parts of the program. At the end of the learning process assignments may rarely contain explanations, while the student has to complete almost all of the program.

During the course of the instruction, the student will perform the instructional tasks as required by the completion assignments, and the success or failure on each completion assignment will repeatedly lead to an update of the student profile. Based on this update, sets of new candidate plans are formed in order to select the following problem from the problem database and to compose the next completion assignment.

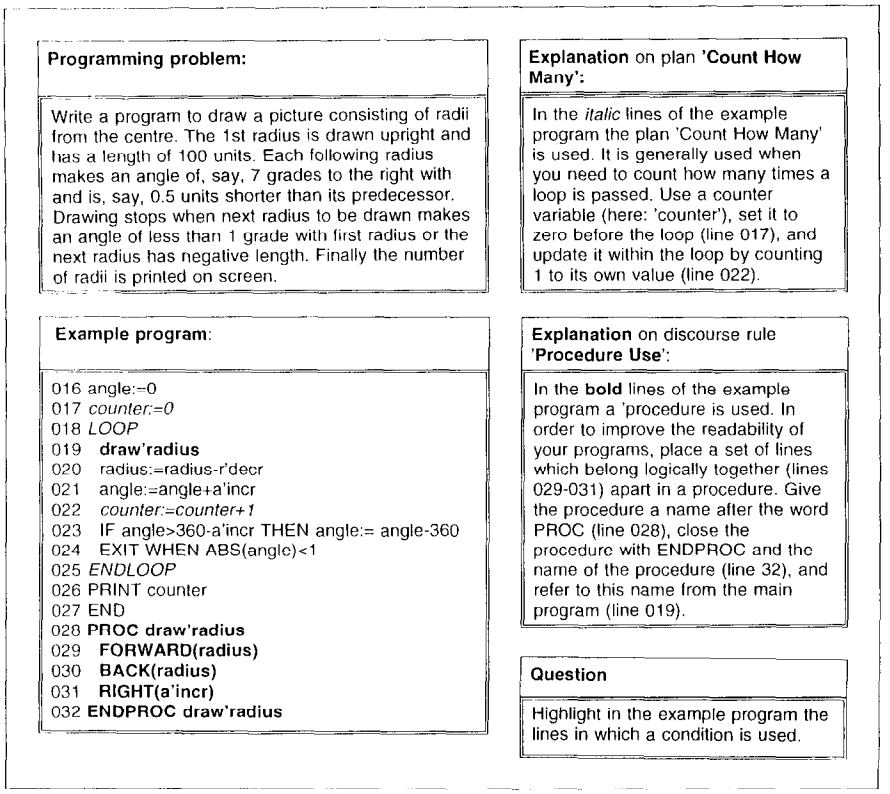

Figure 6. Example assignment with two explanations and one instance question.

DISCUSSION

In this article, a taxonomy of content elements for an introductory programming course was presented. With the content elements being organized according to this taxonomy and suitably stored in the knowledge bases of an ITS, a large variety of deliveries can be composed by the system. The presented approach and its potentials for delivery composition were illustrated for two instructional models: a subset of Merrill's Component Display Theory and Van Merriënboer's Completion Strategy.

As a general goal, the taxonomy of content elements should enable an ITS shell to adapt a variety of instructional models independent of the domain. In addition, it should open the way to implement separate modules for content planning and delivery planning, as illustrated in our description of CASCO. However, some major reservations must be made here.

First, different domains may require different taxonomies of content elements. For this reason, the variability of domains covered in the ITSSEL project is restricted to executable languages, which are assumed to be similar enough to be treated under the same taxonomy.

Second, our taxonomy is based on the problem solving process that a novice programmer should learn. In courses for professional software engineers a much more refined procedure should be taught. This distinction is a consequence of the observation that the same domain may require different representations for novices ("propaedeutic representations"; Halff, 1988) and experts. In our opinion, these dif-

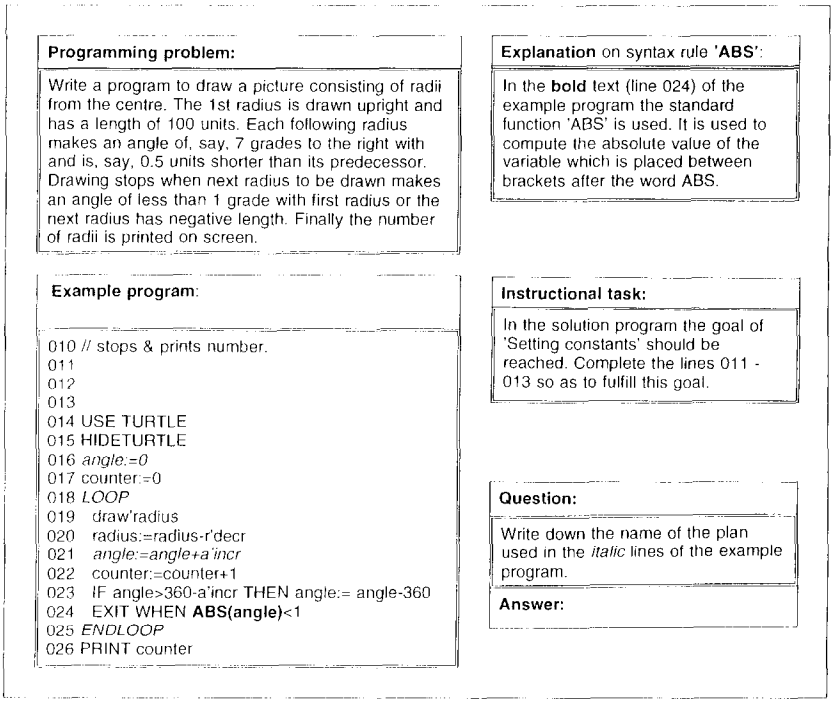

Figure 7. Example assignment with explanation, generality question, and completion task.

ferences will ever form an obstacle to the complete realization of sufficiently powerful domain-independent instructional models.

Future research will be mainly concerned with a further extension of the Completion Strategy. Essentially, three interrelated levels may be distinguished in our taxonomy of content: the goal level, the plan level, and the program code level. On the goal level, one should have knowledge of cues in the problem text, possible programming goals, and analysis heuristics to reach a goal decomposition; on the plan level, one should have knowledge of the programming goals, possible programming plans, and plan principles to reach a plan specification; and on the program code level, one should have knowledge of the programming plans, syntax rules, and discourse rules to reach correct program code. Thus, programming goals are important learning elements for both the goal level and the plan level, and programming plans are important learning elements for both the plan level and the program code level. As described in this article, the Completion Strategy is yet mainly concerned with the program code level; for this reason, the used learning elements are restricted to programming plans, syntax rules, and discourse rules.

However, the basic ideas underlying the Completion Strategy may also be applied to the other two levels. For instance, one may also present the student with incomplete goal decompositions, or incomplete plan specifications, which must be completed. If the Completion Strategy is used on these higher levels, it is required that programming goals, analysis heuristics and plan principles are also presented as learning elements to the students. In addition, an important extension of the Completion Strategy on the program code level may be the inclusion of test heuristics as an extra learning element, because according to several authors (e.g., Pea,

1986; Van Merriënboer & De Croock, in press) the testing and debugging of programs may yield an important contribution to learning outcomes.

Acknowledgments — This research is financed by the Ministry of Education within the framework of the so-called "Technology Renewal Project" of the Dutch technical universities.

REFERENCES

Barr, A., Beard, M., & Atkinson, R. C. (1976). The computer as a tutorial laboratory: The Stanford BIP project. *International Journal of Man-Machine Studies*, **8**, 567–596.

Bonar, J. G., & Cunningham, R. (1988a). *Bridge: Intelligent tutoring with intermediate representations*. Pittsburgh: University of Pittsburgh, Learning Research and Development Center.

Bonar, J. G., & Cunningham, R. (1988b). Bridge: Tutoring the programming process. In J. Psotka, L. D. Massey, & S. A. Mutter (Eds.), *Intelligent tutoring systems: Lessons learned* (pp. 409–434). Hillsdale, NJ: Erlbaum.

Brecht (Wasson), B. J., MacCalla, G. I., Greer, J. E., & Jones, M. (1989). Planning the content of instruction. In D. Bierman, J. Breuker, & J. Sandberg (Eds.), *Artificial intelligence and education: Proceedings of the 4th International Conference on AI and Education* (pp. 32–41). Amsterdam: IOS.

Burns, H. L., & Caps, C. G. (1988). Foundations of intelligent tutoring systems: An introduction. In M. C. Polson & J. J. Richardson (Eds.), *Foundations of intelligent tutoring systems* (pp. 1–19). Hillsdale, NJ: Erlbaum.

Christensen, B. R. (1982). *Beginning COMAL*. Chichester, England: Ellis Horwood.

Gegg-Harrison, T. S. (1991). Learning Prolog in a schema-based environment. *Instructional Science*, **20**, 173–192.

Halff, H. M. (1988). Curriculum and instruction in automated tutors. In M. C. Polson & J. J. Richardson (Eds.), *Foundations of intelligent tutoring systems* (pp. 79–108). Hillsdale, NJ: Erlbaum.

Johnson, W. L., & Soloway, E. (1985). *Micro-PROUST* (Tech. Rep. 402). New Haven, CT: Yale University, Department of Computer Science.

Johnson, W. L., & Soloway, E. (1987). PROUST: An automatic debugger for Pascal programs. In G. Kearsley (Ed.), *Artificial intelligence and instruction: Applications and methods* (pp. 49–67). Reading, MA: Addison-Wesley.

Joni, S. A., & Soloway, E. (1986). But my program runs! Discourse rules for novice programmers. *Journal of Educational Computing Research*, **2**, 95–125.

Krammer, H. P. M. (1990). *Instructional models for ITSSEL* (Memorandum ITSSEL-90-2). Enschede, The Netherlands: University of Twente, Department of Computer Science/Department of Education.

Krammer, H. P. M., & Dijkstra, S. (1992, March). *The automatic sequencing of problems for introductory programming*. Paper presented at the NATO Advanced Research Workshop "Automating Instructional Design, Development, and Delivery," Barcelona, Spain.

Maaswinkel, R. M., & Offereins, M. (1990). *Preliminary design of the architecture of an intelligent tutoring system* (Memorandum ITSSEL-90-4). Enschede, The Netherlands: University of Twente, Department of Computer Science/Department of Education.

Macmillan, S. A., Emme, D., & Berkowitz, M. (1988). Instructional planners: Lessons learned. In J. Psotka, L. D. Massey, & S. A. Mutter (Eds.), *Intelligent tutoring systems: Lessons learned* (pp. 229–256). Hillsdale, NJ: Erlbaum.

Merrill, M. D. (1983). Component display theory. In C. M. Reigeluth (Ed.), *Instructional-design theories and models: An overview of their current status* (pp. 279–333). Hillsdale, NJ: Erlbaum.

Merrill, M. D. (in press). An introduction to instructional transaction theory. In S. Dijkstra, H. P. M. Krammer, & J. J. G. Van Merriënboer (Eds.), *Instructional models in computer-based learning environments*. Heidelberg, Germany: Springer Verlag.

Murray, W. R. (1989). Control for intelligent tutoring systems: A blackboard-based dynamic instructional planner. In D. Bierman, J. Breuker, & J. Sandberg (Eds.), *Artificial intelligence and education: Proceedings of the 4th International Conference on AI and Education* (pp. 150–168). Amsterdam: IOS.

O'Neil, H. F., Slawson, D. A., & Baker, E. L. (1991). Design of a domain-independent problem-solving instructional strategy for intelligent computer-assisted instruction. In H. Burns, J. W. Parlett, &

C. L. Redfield (Eds.), *Intelligent tutoring systems: Evolutions in design* (pp. 69–103). Hillsdale, NJ: Erlbaum.

Park, O.-C., Perez, R. S., & Seidel, R. J. (1987). Intelligent CAI: Old wine in new bottles, or a new vintage?. In G. Kearsley (Ed.), *Artificial intelligence and instruction: Applications and methods* (pp. 11–45). Reading, MA: Addison-Wesley.

Pea, R. D. (1986). Language independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, **2**, 25–36.

Reigeluth, C. M. (1983). Instructional design: What is it and why is it?. In C. M. Reigeluth (Ed.), *Instructional-design theories and models: An overview of their current status* (pp. 3–36). Hillsdale, NJ: Erlbaum.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, **13**, 389–414.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions of Software Engineering*, **SE-10**, 595–609.

Spohrer, J., Soloway, E., & Pope, E. (1985). A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction*, **1**, 163–207.

Van Merriënboer, J. J. G. (1990a). Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of Educational Computing Research*, **6**, 265–285.

Van Merriënboer, J. J. G. (1990b). *Teaching introductory computer programming: A perspective from instructional technology* (Doctoral dissertation, University of Twente). Enschede, The Netherlands: Bijlstra & Van Merriënboer.

Van Merriënboer, J. J. G. (in press). Training strategies for teaching introductory computer programming. In D. G. Bouwhuis (Ed.), *Cognitive modelling and interactive environments*. Heidelberg, Germany: Springer Verlag.

Van Merriënboer, J. J. G., & De Croock, M. B. M. (in press). Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research*.

Van Merriënboer, J. J. G., & Krammer, H. P. M. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science*, **16**, 251–285.

Van Merriënboer, J. J. G., & Krammer, H. P. M. (1989). The "completion strategy" in programming instruction: Theoretical and empirical support. In S. Dijkstra, B. H. A. M. van Hout Wolters, & P. C. van der Sijde (Eds.), *Research on instruction: Design and effects* (pp. 45–61). Englewood Cliffs, NJ: Educational Technology Publications.

Van Merriënboer, J. J. G., Krammer, H. P. M., & Maaswinkel, R. M. (1992, March). *Automating the planning and construction of programming assignments for teaching introductory computer programming*. Paper presented at the NATO Advanced Research Workshop "Automating Instructional Design, Development, and Delivery," Barcelona, Spain.

Van Merriënboer, J. J. G., & Paas, F. G. W. C. (1990). Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior*, **6**, 273–289.

Wasson, B. J. (in press). PEPE: A computational framework for a content planner. In S. Dijkstra, H. P. M. Krammer, & J. J. G. Van Merriënboer (Eds.), *Instructional models in computer-based learning environments*. Heidelberg, Germany: Springer Verlag.

Wiedenbeck, S. (1986). Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, **25**, 697–709.