6

Structured Business Process Specification

Isn't it compelling to apply the structured programming arguments to the field of business process modeling? Our answer to this question is 'no'.

The principle of structured programming emerged in the computer science community. From today's perspective, the discussion of structured programming rather had the characteristics of a maturing process than the characteristics of a debate, although there have also been some prominent skeptical comments on the unrestricted validity of the structured programming principle. Structured programming is a well-established design principle in the field of program design like the third normal form in the field of database design. It is common sense that structured programming is better than unstructured programming – or let's say structurally unrestricted programming – and this is what is taught as foundational knowledge in many standard curricula of many software engineering study programmes. With respect to business process modeling, in practice, you find huge business process models that are arbitrary nets. How come? Is it somehow due to some lack of knowledge transfer from the programming language community to the information systemcommunity For computer scientists, it might be tempting to state that structured programming is a proven concept and it is therefore necessary to eventually promote a structured business process modeling discipline, however, care must be taken.

We want to contribute to the understanding in how far a structured approach can be applied to business process modeling and in how far such an approach is naive [113]. We attempt to clarify that the arguments of structured programming are about the pragmatics of programming. Furthermore, we want to clarify that, in our opinion, argumentations in favor of structured programming often appeal to evidence. Consequentially, our reasoning is at the level of pragmatics of business process modeling. We try to avoid getting lost in superficial comparisons of modeling language constructs but trying to understand the core problems of structuring business process specifications. As an example, so to speak as a taster to our discussion, we take forward one of our arguments here, which is subtle but important, i.e., that there are some

D. Draheim, *Business Process Technology: A Unified View on Business Processes, Workflows and Enterprise Applications*, DOI 10.1007/978-3-642-01588-5_6,

© Springer-Verlag Berlin Heidelberg 2010

diagrams expressing behavior that cannot be transformed into a structured diagram expressing the same behavior solely in terms of the same primitives as the original structurally unrestricted diagram. These are all those diagrams that contain a loop which is exited via more than one path to the end point, which is a known result from literature, encountered [37] by Corrado Böhm and Guiseppe Jacopini, proven for a special case [206] by Donald E. Knuth and Robert W. Floyd and proven in general [208] by S. Rao Kosaraju.

On a first impression, structured programs and flowcharts appear neat and programs and flowcharts with arbitrary jumps appear obfuscated, muddle-headed, spaghetti-like etc. [76]. But the question is not to identify a subset of diagrams and programs that look particularly fine. The question is, given a behavior that needs description, whether it makes always sense to replace a description of this behavior by a new structured description. What efforts are needed to search for a good alternative description? Is the resulting alternative structured description as nice as the original non-structured description?

Furthermore, we need to gain more systematic insight into which metrics we want to use to judge the quality of a description of a behavior, because categories like neatness or prettiness are not satisfactory for this purpose if we are serious that our domain of software development should be oriented rather towards engineering [256, 51] than oriented towards arts and our domain of business management should be oriented rather towards science [144]. Admittedly, however, both fields are currently still in the stage of pre-paradigmatic research [223]. All these issues form the topic of investigation of this chapter.

For us, the definitely working theory of quality of business process models would be strictly pecuniary, i.e., it would enable us to define a style guide for business process modeling that eventually saves costs in system analysis and software engineering projects. The better the cost-savings realized by the application of such a style-guide the better such a theory. Because our ideal is pecuniary, we deal merely with functionality. There is no cover, no aesthetics, no mystics. This means there is no form in the sense of Louis H. Sullivan [332] – just function.

6.1 Basic Definitions

In this section we explain the notions of program, structured program, flowchart, D-flowchart, structured flowchart, business process model and structured business process model as used in this chapter. The focus of this section is on syntactical issues. You might want to skip this section and use it as a reference, however, you should at least glimpse over the formation rules of structured flowcharts defined in Fig. 6.1, which are also the basis for structured business process modeling.

In the course of this chapter, programs are imperative programs which may contain ‘go to’-statements, i.e., they consist of basic statements, sequences, case constructs, loops and ‘go to’-statements. Structured programs are those

programs that abstain from ‘go to’-statements. Loops, i.e., explicit programming constructs for loops, do not add to the expressive power of a programming language with ‘go to’-statements – in presence of ‘go to’-statements loops are syntactic sugar. Flowcharts correspond to programs. Flowcharts are directed graphs with nodes being basic activities, decision points or join points. A directed circle in a flowchart can be interpreted as a loop or as the usage of a ‘go to’-statement. In general flowcharts it allowed to place join points arbitrarily, which makes it possible to create spaghetti structures, i.e., arbitrary jump structures, like the ‘go to’-statements allows for the creation of spaghetti code.

It is a matter of taste whether to make decision and joint points explicit nodes or not. If you strictly use decision and joint points the basic activities always have exactly one incoming and one outgoing edge. In concrete modeling languages like event-driven process chains, there are usually some more constraints, e.g., a constraint on decision points not to have more than one incoming edge or a constraint on join points to have not more than one outgoing edge. If you allow basic activities to have more than one incoming edge you do not need join points any more. Similarly, you can get rid of a decision point by using several outgoing edges by directly connecting the several branches of the decision point as outgoing edges to a basic activity and labeling the several branches with appropriate flow conditions. For example, in formcharts [89] we have chosen the option not to use explicit decision and join points. Our discussion is independent from the detail question of having explicit or implicit decision and join points, because both concepts are interchangeable. Therefore, we feel free to use both options.

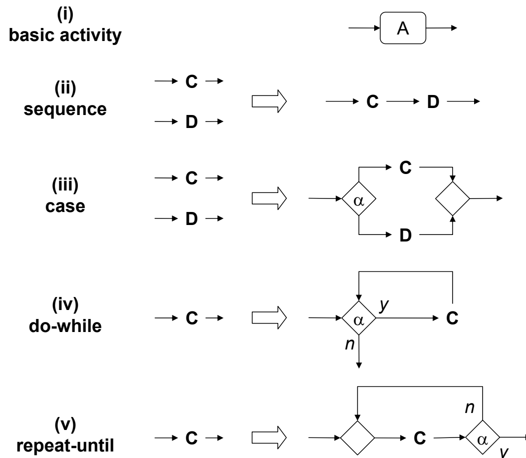

Fig. 6.1. Semi-formal formation rules for structured flowcharts.

6.1.1 D-Charts

It is possible to define formation rules for a restricted class of flowcharts that correspond to structured programs. In [208] these diagrams are called Dijkstra-flowcharts or D-flowcharts for short, named after Edgser W. Dijkstra. Figure 6.1 summarizes the semi-formal formation rules for D-flowcharts.

Actually, the original definition of D-flowcharts in [208] consists of the formation rules (i) to (iv) with one formation rule for each programming language construct of a minimal structured imperative programming language with basic statements, sequences, case-constructs and while-loops with basic activities in the flowchart corresponding to basic statements in the programming language. We have added a formation rule (v) for the representation of repeat-until-loops and call flowcharts resulting from rules (i) to (v) structured flowcharts in the sequel.

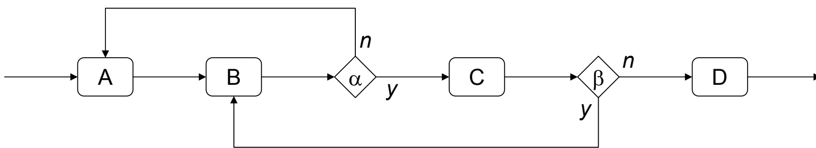

Fig. 6.2. Example flowchart that is not a D-flowchart.

The flowchart in Fig. 6.2 is not a structured flowchart, i.e., it cannot be derived from the formation rules in Fig. 6.1. The flowchart in Fig. 6.2 can be interpreted as consisting of a repeat-until-loop exited via the α-decision point and followed by further activities C and D. In this case, the β-decision point can lead to a branch that jumps into the repeat-until-loop in addition to the regular loop entry point via activity A, which infringes the structured programming and structured modeling principle and gives raises to spaghetti structure. Thus, the flowchart in Fig. 6.2 visualizes the program in Listing 6.1

Listing 6.1 Textual presentation of the business process in Fig. 6.2 with a jump into the loop.

```
01 REPEAT
02   A;
03   B;
04 UNTIL alpha;
05 C;
06 IF beta THEN GOTO 03;
07 D;
```

The flowchart in Fig. 6.2 can also be interpreted as consisting of a while-loop exited via the β-decision point, where the while-loop is surrounded by a preceding activity A and a succeeding activity D. In this case, the α-decision point can lead to a branch that jumps out of the while-loop in addition to the regular loop exit via the β-decision point, which again infringes the structured modeling principle. Thus, the flowchart in Fig. 6.2 also visualizes the program in Listing 6.2

Listing 6.2 Alternative textual presentation of the business process in Fig. 6.2 with a jump out of the loop.

```
01 A;
02 REPEAT
03   B;
04   IF NOT alpha THEN GOTO 01
05   C;
06 UNTIL NOT beta;
07 D;
```

Flowcharts are visualization of programs. In general, a flowchart can be interpreted ambiguously as the visualization of several different program texts, because, for example, on the one hand, an edge from a decision point to a join point can be interpreted as a 'go to'-statement or, on the other hand, as the back branch from an exit point of a repeat-until loop to the start of the loop. Structured flowcharts are visualizations of structured programs. Loops in structured programs and structured flowcharts enjoy the property that they have exactly one entry point and exactly one exit point. Whereas the entry point and the exit point of a repeat-until loop are different, the entry point and exit point of a while-loop are the same, so that a while-loop in a structured flowchart has exactly one contact point. That might be the reason that structured flowcharts that use only while-loops instead of repeat-until loops appear more normalized. Similarly, in a structured program and flowchart all case-constructs has exactly one entry point and one exit point. In general, additional entry and exit points can be added to loops and case constructs by the usage of 'go to'-statements in programs and by the usage of arbitrary decision points in flowcharts. In structured flowcharts, decision points are introduced as part of the loop constructs and part of the case construct. In structured programs and flowcharts, loops and case-constructs are strictly nested along the lines of the derivation of their abstract syntax tree.

Business process models extend flowcharts with further modeling elements like a parallel split, parallel join or non-deterministic choice. Basically, we discuss the issue of structuring business process models in terms of flowcharts, because flowcharts actually are business process model diagrams, i.e., flowcharts

form a subset of business process models. As the constructs in the formation rules of Fig. 6.1 further business process modeling elements can also be introduced in a structured manner with the result of having again only such diagrams that are strictly nested in terms of their looping and branching constructs. For example, in such a definition the parallel split and the parallel join would not be introduced separately but as belonging to a parallel modeling construct.

6.1.2 A Notion of Equivalence for Business Processes

Bisimilarity has been defined formally in [273] as an equivalence relation for infinite automaton behavior, i.e., process algebra [249, 250]. Bisimilarity expresses that two processes are equal in terms of their observable behavior. Observable behavior is the appropriate notion for the comparison of automatic processes. The semantics of a process can also be understood as opportunities of one process interacting with another process. Observable behavior and experienced opportunities are different viewpoints on the semantics of a process, however, whichever viewpoint is chosen, it does not change the basic concept of bisimilarity. Business processes can be fully automatic; however, business processes can also be descriptions of human actions and therefore can also be rather a protocol of possible steps undertaken by a human. We therefore choose to explain bisimilarity in terms of opportunities of an actor, or, as a metaphor, from the perspective of a player that uses the process description as a game board – which neatly fits to the notions of simulation and bisimulation, i.e., bisimilarity.

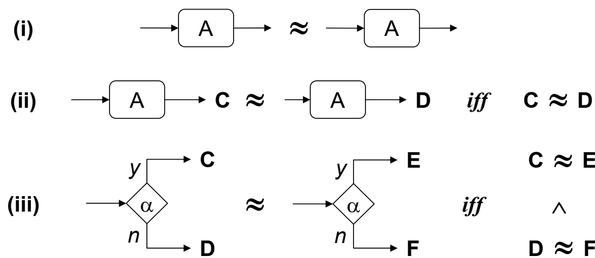

Fig. 6.3. Characterization of bisimilarity for business process models.

In general, two processes are bisimilar if starting from the start node they reveal the same opportunities and each pair of same opportunities lead again to bisimilar processes. More formally, bisimilarity is defined on labeled transition systems [3] as the existence of a bisimulation, which is a relationship that enjoys the aforementioned property, i.e., nodes related by the bisimilarity lead via the same opportunities to nodes that are related again, i.e., recursively, by the bisimilarity. In the non-structured models the opportunities are edges

leading out of an activity and the two edges leading out of a decision point. For our purposes, bisimilarity can be characterized by the rules in Fig. 6.3.

6.2 The Pragmatics of Structuring Business Processes

6.2.1 Resolving Arbitrary Jump Structures

Have a look at Fig. 6.4. Like Fig. 6.2 it shows a business process model that is not a structured business process model. The business process described by the business process model in Fig. 6.4 can also be described in the style of a program text as in Listing 6.3.

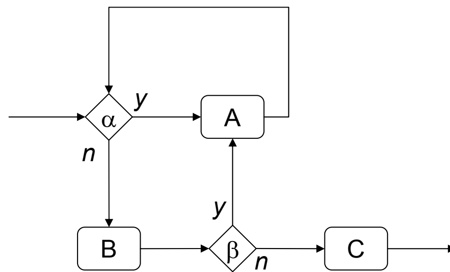

Fig. 6.4. Example business process model that is not structured.

In textual representation or interpretation in Fig. 6.4, the business process model in Fig. 6.4 consists of a while-loop followed by a further activity B, a decision point that might branch back into the while-loop and eventually an activity C. Alternatively, the business process can also be described by structured business process models. Fig. 6.5 shows two examples of such structured business process models and Listings 6.4 and 6.5 show the corresponding program text representations that are visualized by the business process models in Fig. 6.5.

Listing 6.3 Textual presentation of the business process in Fig. 6.4.

```
01 WHILE alpha DO
02   A;
03 B;
04 IF beta THEN GOTO 02;
05 C;
```

The business process models in Figs. 6.4 and 6.5 resp. Listings 6.3, 6.4 and 6.5 describe the same business process. They describe the same business

process, because they are bisimilar, i.e., in terms of their nodes, which are, basically, activities and decision points, they describe the same observable behavior resp. same opportunities to act for an actor – we have explained the notion of equality and the more precise approach of bisimilarity in more detail in Sect. 6.1.2.

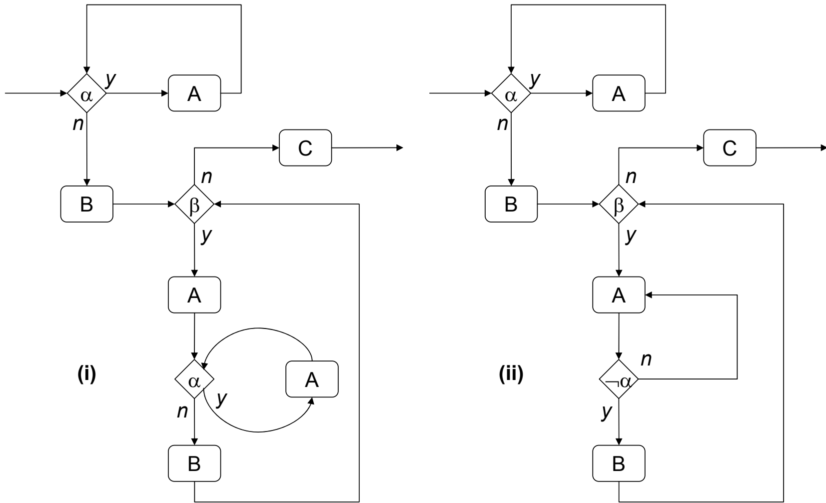

Fig. 6.5. Structured business process models that replace the non-structured one in Fig. 6.4.

The derivation of the business process models in Fig. 6.5 from the formation rules given in Fig. 6.1 can be understood by looking at its abstract syntax tree, which appears at tree ψ in Fig. 6.6. The proof that the process models in Figs. 6.4 and 6.5 are bisimilar is left to the reader as an exercise. The reader is also invited to find structured business process models that are less complex than the ones given in Fig. 6.5, whereas complexity is an informal concept that depends heavily on the perception and opinion of the modeler. For example, the model (ii) in Fig. 6.4 results from an immediate simple attempt to reduce the complexity of the model (i) in Fig. 6.5 by eliminating the A-activity which follows the α-decision point and connecting the succeeding 'yes'-branch of the α-decision point directly back with the A-activity preceding the decision point, i.e., by reducing a while-loop-construct with a preceding statement to a repeat-until-construct. Note, that the model in Fig. 6.5 has been derived from the model in Fig. 6.4 by straightforwardly unfolding it behind the β-decision point as much as necessary to yield a structured description of the business process. In what sense the transformation from model (i) to model (ii) in Fig. 6.5 has lowered complexity and whether it actually or rather superficially has lowered the complexity will be discussed in due course. We will

also discuss another structured business process model with auxiliary logic that is oriented towards identifying repeat-until-loops in the original process descriptions.

Listing 6.4 Textual presentation of business process (i) in Fig. 6.4.

```
01 WHILE alpha DO
02   A;
03 B;
04 WHILE beta DO BEGIN
05   A;
06   WHILE alpha DO
07     A;
08   B;
09 END;
10 C;
```

Listing 6.5 Textual presentation of business process (ii) in Fig. 6.4.

```
01 WHILE alpha DO
02   A;
03 B;
04 WHILE beta DO BEGIN
05   REPEAT
06     A;
07   UNTIL NOT alpha;
08   B;
09 END;
10 C;
```

The above remark on the vagueness of the notion of complexity is not just a side-remark or disclaimer but is at the core of the discussion. If the complexity of a model is a cognitive issue it would be a straightforward approach to let people vote which of the models is more complex. If there is a sufficiently precise method to test whether a person has understood the semantics of a process specification, this method can be exploited in testing groups of people that have been given different kinds of specifications of the same process and concluding from the test results which of the process specifications should be considered more complex. Such an approach relies on the preciseness of the semantics and eventually on the quality of the test method. We will suggest to consider such a test method approach again in Sect. 6.3 in the discussion of structured programming, because program have a definite semantics as functional transforms.

It is a real challenge to search for a definition of complexity of models or their representations. What we expect is that less complexity has something to do with better quality, and before we undertake efforts in defining complexity of models we should first understand possibilities to measure the quality of models. The usual measures by which modelers and programmers often judge complexity of models like understandability or readability are vague concepts themselves. Other categories like maintainability or reusability are more concrete than understandability or readability but still vague. Of course, we can define metrics for the complexity of diagrams. For example, it is possible to define that the number of activity nodes used in a business process model increases the complexity of a model. The problem with such metrics is that it follows immediately that the model in Fig. 6.5 is more complex than the model in Fig. 6.4. Actually, this is what we believe.

6.2.2 Immediate Arguments For and Against Structure

We believe that the models in Fig. 6.5 are more complex than the models in Fig. 6.4. A structured approach to business process models would make us believe that structured models are somehow better than non-structured models in the same way that the structured programming approach believes that structured programs are somehow better than non-structured programs. So either less complexity must not always be better or the tenets of the structured approach must be loosened to a rule of thumb, i.e., the belief that structured models are in general better than non-structured models, despite some exceptions like our current example. An argument in favor of the structured approach could be that our current example is simply too small, i.e., that the aforementioned exceptions are made of small models or, to say it differently, that the arguments of a structured approach become valid for models beyond a certain size. We do not think so. We rather believe that our discussion scales, i.e., that the arguments that we will give below also apply equally and even more so for larger models. We want to approach these questions more systematically.

In order to do so, we need to answer why we believe that the models in Fig. 6.5 are more complex than the model in Fig. 6.4. The immediate answer is simply because they are larger and therefore harder to grasp, i.e., a very direct cognitive argument. But there is another important argument why we believe this. The model in Fig. 6.4 shows an internal reuse that the models in Fig. 6.5 do not show. The crucial point is the reuse of the loop consisting of the A-activity and the α-decision point in Fig. 6.4. We need to delve into this important aspect and will actually do this later. First, we want to discuss the dual question, which is of equal importance, i.e., we must also try to understand or try to answer the question, why modelers and programmers might find that the models in Fig. 6.5 are less complex than the models in Fig. 6.4.

A standard answer to this latter question could typically be that the edge from the β-decision point to the A-activity in Fig. 6.4 is an arbitrary jump, i.e., a spaghetti, whereas the diagrams in Fig. 6.5 do not show any arbitrary jumps or spaghetti-like phenomena. But the question is whether this vague argument can be made more precise. A structured diagram consists of strictly nested blocks. All blocks of a structured diagram form a tree-like structure according to their nesting, which corresponds also to the derivation tree in terms of the formation rules of Fig. 6.1. The crucial point is that each block can be considered a semantic capsule from the viewpoint of its context. This means, that once the semantics of a block is understood by the analyst studying the model, the analyst can forget about the inner modeling elements of the block. This is not so for diagrams in general. This has been the argument of looking from outside onto a block in the case a modeler want to know its semantics in order to understand the semantics of the context where it is utilized. Also, the dual scenario can be convincing. If an analyst is interested in understanding the semantics of a block he can do this in terms of the inner elements of a block only. Once the analyst has identified the block he can forget about its context to understand it. This is not so easy in a non-structured language. When passing an element, in general you do not know where you end up in following the various paths behind it. It is also possible to subdivide a non-structured diagram into chunks that are smaller than the original diagram and that make sense to understand as capsules. For example, this can be done, if possible, by transforming the diagram into a structured one, in which you will find regions of your original diagram. However, it is extra effort to do this partition.

With the current set of modeling elements, i.e., those introduced by the formulation rules in Fig. 6.1, all this can be seen particularly easy, because each block has exactly one entry point, i.e., one edge leading into it. Fortunately, standard building blocks found in process modeling would have one entry point in a structured approach. If you have, in general, also blocks with more than one entry points, it would make the discussion interesting. The above argument would not be completely infringed. Blocks still are capsules, with a semantics that can be understood locally with respect to their appearance in a strictly nested structure of blocks. The scenario itself remains neat and tidy; the difference lies in the fact that a block with more than one entry has a particular complex semantics in a certain sense. The semantics of a block with more than one entry is manifold, e.g., the semantics of a block with two entries is threefold. Given that, in general, we also have concurrency phenomena in a business process model, the semantics of block with two entry points, i.e., its behavior or opportunities, must be understood for the case that the block is entered via one or the other entry point and for the case that the block is entered simultaneously. But this is actually not a problem; it just means a more sophisticated semantics and more documentation.

Despite a more complex semantics, a block with multiple entries still remains an anchor in the process of understanding a business process model,

because it is possible, e.g., to understand the model from inside to outside following the strict tree-like nesting, which is a canonical way to understand the diagram, i.e., a way that is always defined. It is also always possible to understand the diagram sequentially from the start node to the end node in a controlled manner. The case constructs make such sequential proceeding complex, because they open alternative paths in a tree-like manner. The advantage of a structured diagram with respect to case-constructs is that each of the alternative paths that are spawned is again a block and it is therefore possible to understand its semantics isolated from the other paths. This is not so in a non-structured diagram, in which there might be arbitrary jumps between the alternative paths, in general. Similarly, if analyzing a structured diagram in a sequential manner, you do not get into arbitrary loops and therefore have to deal with a minimized risk to loose track.

The discussion of the possibility to have blocks with more entry points immediately reminds us of the discussion we have seen within the business process community on multiple versus unique entry points for business processes in a setting of hierarchical decomposition. The relationship between blocks in a flat structured language and sub diagrams in a hierarchical approach and how they play together in a structured approach is an important strand of discussion that we will come back to in due course. For the time being, we just want to point out the relationship between the discussion we just had on blocks with multiple entries and sub diagrams with multiple entries. A counter-argument against sub diagrams with multiple entries would be that they are more complex. Opponents of the argument would say that it is not a real argument, because the complexity of the semantics, i.e., its aforementioned manifoldness, must be described anyhow.

With sub diagrams that may have no more than one entry point, you would need to introduce a manifoldness of diagrams each with a single entry point. We do not discuss here how to transform a given diagram with multiple entries into a manifoldness of diagrams – all we want to remark here that it easily becomes complicated because of the necessity to appropriately handle the aforementioned possibly existing concurrency phenomena. Eventually it turns out to be a problem of transforming the diagram together with its context, i.e., transforming a set of diagrams and sub diagrams with possibly multiple entry points into another set of diagrams and sub diagrams with only unique entry points. Defenders of diagrams with unique entry points would state that it is better to have a manifoldness of such diagrams instead of having a diagram with multiple entries, because, the manifoldness of diagrams documents better the complexity of the semantics of the modeled scenario.

For a better comparison of the discussed models against the above statements we have repainted the diagram from Fig. 6.4 and diagram (ii) from Fig. 6.5 with the blocks they are made of and their abstract syntax trees resp. quasi-abstract syntax tree in Fig. 6.6. The diagram of Fig. 6.4 appears to the left in Fig. 6.6 as diagram Φ and diagram (ii) from Fig. 6.5 appears to the right as diagram Ψ. According to that, the left abstract syntax tree ϕ in

Fig. 6.6 corresponds to the diagram from Fig. 6.4 and the right abstract syntax tree ψ corresponds to the diagram (ii) from Fig. 6.5. Blocks are surrounded by dashed lines in Fig. 6.6.

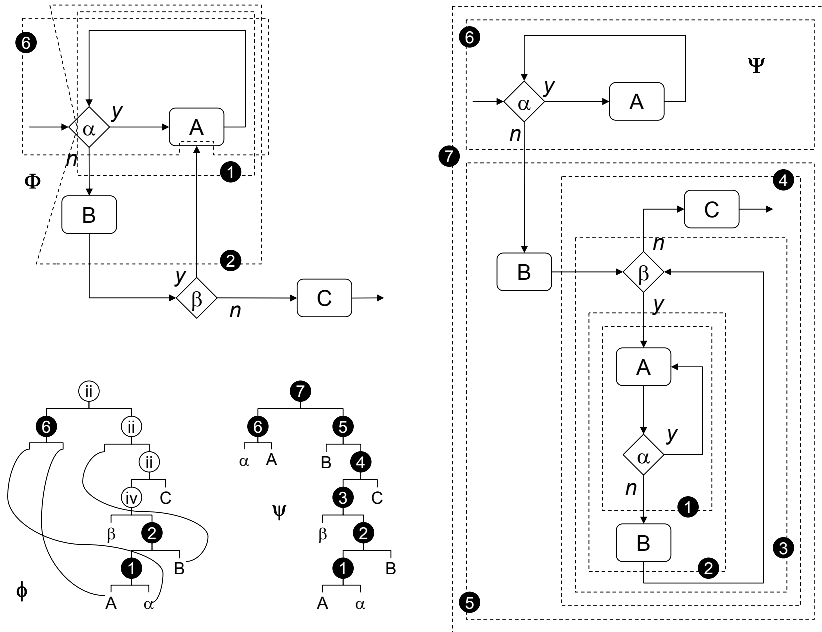

Fig. 6.6. Block-structured versus arbitrary business process model.

If you proceed in understanding the model Φ in Fig. 6.6 you first have to understand a while-loop that encompasses the A-activity – the block labeled with number ‘5’ in model Φ. After that, you are not done with that part of the model. Later, after the β-decision point you are branched back to the A-activity and you have to re-understand the loop it belongs to again, however, this time in a different manner, i.e., as a repeat-until loop – the block labeled with number ‘1’ in model Φ. It is possible to argue that, in some sense, this makes the model Φ harder to read than model Ψ. To say it differently, it is possible to view model Ψ as an instruction manual on how to read the model Φ. Actually, model Ψ is a bloated version of model Φ. It contains some modeling elements of model Φ redundantly, however, it enjoys the property that each modeling element has to be understood only in the context of one block and its encompassing blocks. We can restate these arguments a bit more formally in analyzing the abstract syntax trees ϕ and ψ in Fig. 6.6. Blocks in Ψ correspond to constructs that can be generated by the formation rules in Fig. 6.1. The abstract syntax tree ψ is an alternate presentation of the nesting of blocks in model Ψ. A node stands for a block and for the corresponding

construct according to the formation rules. The graphical model Φ cannot be derived from the formation rules in Fig. 6.1. Therefore it does not possess an abstract syntax tree in which each node represent a unique graphical block and a construct the same time. The tree ϕ shows the problem. You can match the region labeled '1' in model Φ as a block against while-loop-rule (iv) and you can subsequently match the region labeled '2' against the sequence-rule (iii). But then you get stuck. You can form a further do-while loop with rule (iv) out of the β-decision point and block '2' as in model Ψ but the resulting graphical model cannot be interpreted as a part of model Φ any more. This is because the edge from activity B to the β-decision point graphically serves both as input branch to the decision point and as back branch to the decision point. This graphical problem is resolved in the abstract syntax tree ϕ by reusing the activity B in the node that corresponds to node '5' in tree ψ in forming a sequence according to rule (ii) with the results that the tree ϕ is actually no longer a tree. Similarly, the reuse of the modeling elements in forming node '6' in the abstract syntax tree ϕ visualizes the double interpretation of this graphical region as both a do-while loop and repeat-until loop.

6.2.3 Structure for Text-based versus Graphical Specifications

In Sect. 6.2.2 we have said that an argument for a structured business process specification is that it is made of strictly nested blocks and that each identifiable block forms a semantic capsule. In the argumentation we have looked at the graphical presentation of the models only and now we will have a look also at the textual representations.

This section needs a disclaimer. We are convinced that it is risky in the discussion of quality of models to give arguments in terms of cognitive categories like understandability, readability, cleanness, well-designedness and well-definedness. These categories tend to have an insufficient degree of definedness themselves so that argumentations based on them easily suffer a lack of falsifiability. Nevertheless, in this section, in order to abbreviate, we need to speak directly about the reading ease of specifications. The judgments are our very own opinion, an opinion that expresses our perception of certain specifications. The reader may have a different opinion and this would be interesting in its own right. At least, the expression of our own opinion may encourage the reader to judge about the readability certain specifications.

As we said in terms of complexity, we think that the model in Fig. 6.4 is easier to understand than the models in Fig. 6.5. We think it is easier to grasp. Somehow paradoxically, we think the opposite about the respective text representation, at least at a first sight, i.e., as long as we have not internalized too much all the different graphical models in listings. This means, we think that the text representation of the models in Fig. 6.4, i.e., Listing 6.3, is definitely harder to understand than the text representation of both models in Fig. 6.5, i.e., Listings 6.4 and 6.5. How comes? Maybe, the following observation helps, i.e., that we also think that the graphical model in Fig. 6.5 is also easier to

read than the model's textual representation in Listing 6.3 and also easier to read than the two other Listings 6.4 and 6.5. Why is Listing 6.5 so relatively hard to understand? We think, because there is no explicitly visible connecting between the jumping-off point in line '04' and the jumping target in line line '02'. Actually, the first thing we would recommend in order to understand Listing 6.5 better is to draw its visualization, i.e., the model in Fig. 6.5, or to concentrate and to visualize it in our mind. By the way, we think that drawing some arrows in Listing 6.3 as we did in Fig. 6.7 also help. The two arrows already help despite the fact that they make explicit only a part of the jump structure – one possible jump from line '01' to line '03' in case the α-condition becomes invalid must still be understood by the indentation of the text.

```
01 WHILE alpha DO
02   A;
03 B;
04 IF beta THEN GOTO 02;
05 C;
```

Fig. 6.7. Listing enriched with arrows for making jump structure explicit.

All this is said for such a small model consisting of a total of five lines. Imagine, if you had to deal with a model consisting of several hundreds lines with arbitrary 'go to'-statements all over the text. If it is true that the model in Fig. 6.4 is easier to understand than the models in Fig. 6.5 and at the same time Listing 6.3 is harder to understand than Listings 6.4 and 6.5 this may lead us to the assumption that the understandability of graphically presented models follows other rules than the understandability of textual representation. Reasons for this may be, on the one hand, the aforementioned lack of explicit visualizations of jumps, and, on the other hand, the one-dimensional layout of textual representations. The reason why we have given all of these arguments in this section is not in order to promote visual modeling. The reason is that we see a chance that they might explain why the structural approach has been so easily adopted in the field of programming.

The field of programming was and still is dominated by text-based specifications – despite the fact that we have seen many initiatives from syntax-directed editors through to computer-aided software engineering to model-driven architecture. It is fair to remark that the crucial characteristics of mere textual specification in the discussion of this section, i.e., lack of explicit visualization of jumps, or, to say it in a more general manner, support for the understanding of jumps, is actually addressed in professional coding tools like integrated development environments with their maintenance of links, code analyzers and profiling tools. The mere text-orientation of specification has been partly overcome by today's integrated development environments. Let us express once more that we are not promoters of visual modeling or even visual

programming. In [89] we have de-emphasized visual modeling. We strictly believe that visualizations add value, in particular, if it is combined with visual meta-modeling [159, 160, 115]. But we also believe that mere visual specification is no silver bullet, in particular, because it does not scale. We believe in the future of a syntax-direct abstract platform with visualization capabilities that overcomes the gap between modeling and programming from the outset as proposed by the work on AP1 [226, 227] of the Software Engineering research group at the University of Auckland.

6.2.4 Structure and Decomposition

The models in Fig. 6.5 are unfolded versions of the model in Fig. 6.4. Some modeling elements of the diagram in Fig. 6.5 occur redundantly in each model in Fig. 6.4. Such unfolding violates the reuse principle. Let us concentrate on the comparison of the model in Fig. 6.5 with model (i) in Fig. 6.5. The arguments are similar for diagram (ii) in Fig. 6.5. The loop made of the α-decision point and the activity A occurs twice in model (i). In the model in Fig. 6.5 this loop is reused by the jump from the β-decision point albeit via an auxiliary entry point. It is important to understand that reuse is not about the cost-savings of avoiding the repainting of modeling elements but about increasing maintainability.

Imagine, in the lifecycle of the business process a change to the loop consisting of the activity A and the α-decision point becomes necessary. Such changes could be the change of the condition to another one, the change of the activity A to another one or the refinement of the loop, e.g., the insertion of a further activity into it. Imagine that you encounter the necessity for changes by reviewing the start of the business process. In analyzing the diagram, you know that the loop structure is not only used at the beginning of the business process but also later by a possible jump from the β-decision point to it. You will now further analyze whether the necessary changes are only appropriate at the beginning of the business process or also later when the loop is reused from other parts of the business process. In the latter case you are done. This is the point where you can get into trouble with the other version of the business process specification as diagram (i) in Fig. 6.5. You can more easily overlook that the loop is used twice in the diagram; this is particularly true for similar examples in larger or even distributed models. So, you should have extra documentation for the several occurrences of the loop in the process. Even in the case that the changes are relevant only at the beginning of the process you would like to review this fact and investigate whether the changes are relevant for other parts of the process.

It is fair to remark, that in the case that the changes to the loop in question are only relevant to the beginning of the process, the diagram in Fig. 6.5 bears the risk that this leads to an invalid model if the analyst oversees its reuse from later stages in the process, whereas the model (i) in Fig. 6.5 does not bear that risk. But we think this kind of weird fail-safeness can hardly be

sold as an advantage of model (i) in Fig. 6.5. Furthermore, it is also fair to remark, that the documentation of multiple occurrences of a model part can be replaced by appropriate tool-support or methodology like a pattern search feature or hierarchical decomposition as we will discuss in due course. All this amounts to saying that maintainability of a model cannot be reduced to its presentation but depends on a consistent combination of presentational issues, appropriate tool support and defined maintenance policies and guidelines in the framework of a mature change management process.

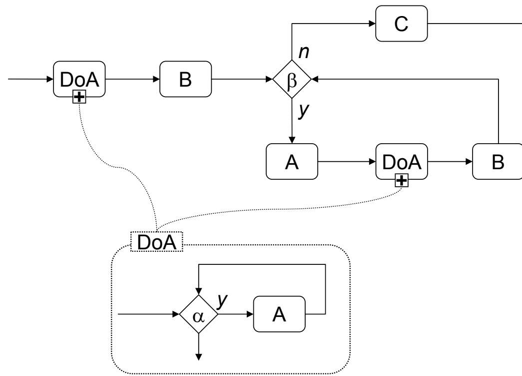

Fig. 6.8. Example business process hierarchy.

We now turn the reused loop consisting of the activity A and the α-decision point in Fig. 6.5 into its own sub diagram in the sense of hierarchical decomposition, give it a name – let us say 'DoA' – and replace the relevant regions in diagram (i) in Fig. 6.5 by the respective, expandable sub diagram activity. The result is shown in Fig. 6.8. Now, it is possible to state that this solution combines the advantages of both kinds of models in question, i.e., it consists of structured models at all levels of the hierarchy and offers an explicit means of documentation of the places of reuse. But caution is necessary. First, the solution does not free the analyst to actually have a look at all the places a diagram is used after he or she has made a change to the model, i.e., an elaborate change policy is still needed. In the small toy example, such checking is easy, but in a tool you usually do not see all sub diagrams at once, but rather step through the levels of the hierarchy and the sub diagrams with links. Remember that the usual motivation to introduce hierarchical decomposition and tool-support for hierarchical decomposition is the desire to deal with the complexity of large and very large models. Second, the tool should not only support the reuse-direction but should also support the inverse use-direction, i.e., it should support the analyst with a report feature that lists all places of reuse for a given sub diagram.

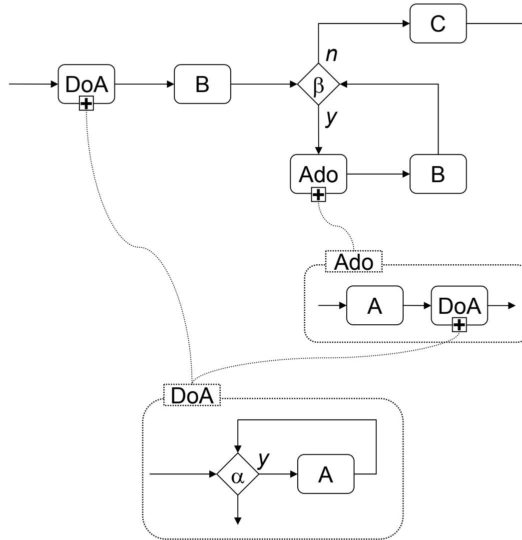

Fig. 6.9. Example for a deeper business process hierarchy.

Now let us turn to a comparative analysis of the complexity of the modeling solution in Fig. 6.8 and the model in Fig. 6.5. The complexity of the top-level diagram in the model hierarchy in Fig. 6.8 is no longer significantly higher than the one of the model in Fig. 6.5. However, together with the sub diagram, the modeling solution in Fig. 6.8 again shows a certain complexity. It would be possible to neglect a reduction of complexity by the solution in Fig. 6.8 completely with the hint that the disappearance of the edge representing the jump from the β-decision point into the loop in Fig. 6.5 is bought by another complex construct in Fig. 6.8 – the dashed line from the activity 'DoA' to the targeted sub diagram. The jump itself can still be seen in Fig. 6.8, somehow, unchanged as an edge from the β-decision point to the activity A. We do not think so. The advantage of the diagram in Fig. 6.8 is that the semantic capsule made of the loop in question is already made explicit as a named sub diagram, which means added documentation value.

Also, have a look at Fig. 6.9. Here the above explanations are even more substantive. The top-level diagram is even less complex than the top-level diagram in Fig. 6.8, because the activity A now has moved to its own level of the hierarchy. However, this comes at the price that now the jump from the β-decision point to the activity A in Fig. 6.5 now re-appears in Fig. 6.9 as the concatenation of the 'yes'-branch in the top-level diagram, the dashed line leading from the activity 'Ado' to the corresponding sub diagram at the next level and the entry edge of this sub diagram.

6.2.5 Business Domain-Oriented versus Documentation-Oriented Modeling

In Sects. 6.2.1 through 6.2.4 we have discussed structured business process modeling for those processes that actually have a structured process specification in terms of a chosen fixed set of activities. In this section we will learn about processes that do not have a structured process specification in that sense. In the running example of Sects. 6.2.1 through 6.2.4 the fixed set of activities was given by the activities of the initial model in Fig. 6.4 and again we will explain the modeling challenge addressed in this section as a model transformation problem.

First, as a further example and for the sake of completeness, we give the resolution of the model in Fig 6.2 into a structured equivalent in Fig. 6.10.

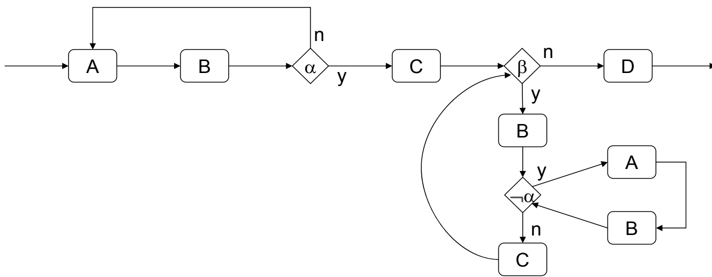

Fig. 6.10. Structured business process model that replaces the non-structured one in Fig. 6.2.

Consider the example business process models in Fig. 6.11. Each model contains a loop with two exits to paths that lead to the end node without the opportunity to come back to the originating loop before reaching the end state. It is known [37, 205, 206, 208] that the behaviors of such loops cannot be expressed in a structured manner, i.e., by a D-chart as defined in Fig. 6.1 solely in terms of the same primitive activities as those occurring in the loop. Extra logic is needed to formulate an alternative, structured specification. Fig. 6.12 shows this loop-pattern abstractly and we proceed to discuss this issues with respect to this abstract model.

Assume that there is a need to model the behavior of a business process in terms of a certain fixed set of activities, i.e., the activities A through D in Fig. 6.12. For example, assume that they are taken from an accepted terminology of a concrete business domain. Other reasons could be that the activities stem from existing contract or service level agreement documents. You can also assume that they are simply the natural choice as primitives for the considered work to be done. We do not delve here into the issue of natural choice and just take for granted that it is the task to model the observed or desired behavior in terms of these activities. For example, we could imagine

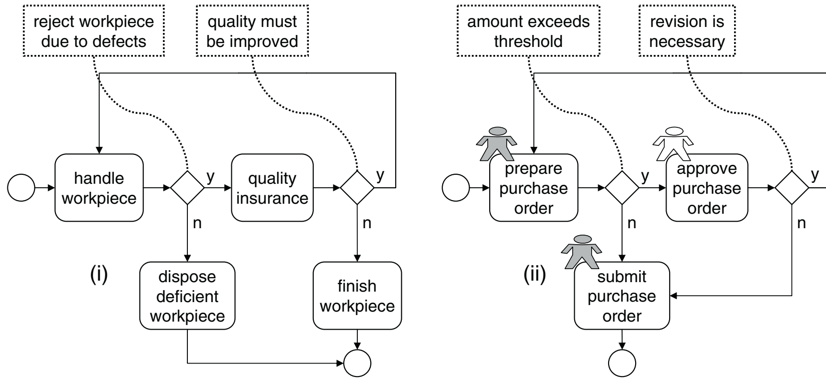

Fig. 6.11. Two example business processes without structured presentation using no other than their own primitives.

an appropriate notion of cohesion of more basic activities that the primitives we are restricted to, or let's say self-restricted to, adhere to. Actually, as it will turn out, for our argumentation to be conclusive there is no need for an explanation how a concrete fixed set of activities arises. What we need for our current argumentation to be conclusive is to demand that the activities are only about actions and objects that are relevant in the business process.

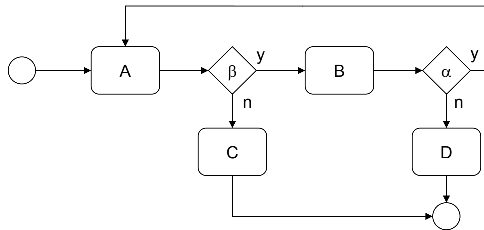

Fig. 6.12. Business process with cycle that is exited via two distinguishable paths.

Fig. 6.13 shows a structured business process model that is intended to describe the same process as the specification in 6.12. In a certain sense it fails. The extra logic introduced in order to get the specification into a structured shape do not belong to the business process that the specification aims to describe. The model in Fig. 6.13 introduces some extra state, i.e., the Boolean variable δ, extra activities to set this variable so that it gets the desired steering effect and an extra δ-decision point. Furthermore, the original δ-decision point in the model of Fig. 6.12 has been changed to a new $\beta \wedge \delta$-decision point. Actually, the restriction of the business process described by Fig. 6.12 onto those particles used in the model in Fig. 6.12 is bisimilar to this process. The problem is that the model in Fig. 6.13 is a hybrid. It is not only a business

domain-oriented model any more, it now has also some merely documentation-related parts. The extra logic and state only serve the purpose to get the diagram into shape. It needs clarification of the semantics. Obviously, it is not intended to change the business process. If the auxiliary introduced state and logic would be also about the business process, this would mean, for example, that in the workshop a mechanism is introduced, for example a machine or a human actor that is henceforth responsible for tracking and monitoring a piece of information δ. So, at least what we need is to explicitly distinguish those elements in such a hybrid model. The question is whether the extra complexity of a hybrid domain- and documentation-oriented modeling approach is justified by the result of having a structured specification.

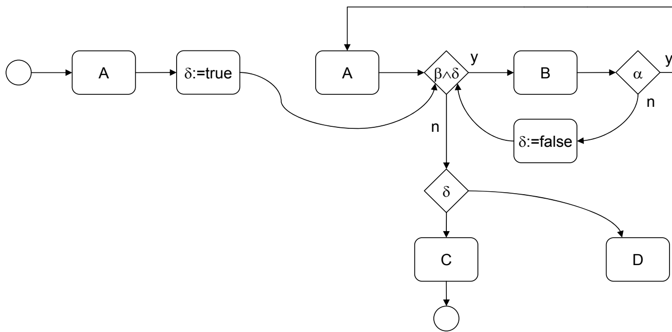

Fig. 6.13. Resolution of business process cycles with multiple distinguishable exits by the usage of auxiliary logic and state.

6.3 Structured Programming

Structured programming is about the design of algorithms. It is about looking for an alternative algorithm that has a somehow better design than a given algorithm but has the same effect. Usually, in the discussion of structured programming the considered effect of algorithms is a functional transformation, i.e., the computation of some output values from some input values. If the effect of a considered algorithm is the behavior of a reactive system things become significantly more complex and the argumentation becomes harder.

6.3.1 An Example Comparison of Program Texts

Let us have a look at an example program of Donald E. Knuth and Robert W. Floyd in [204]. The program is given in Listing 6.6. The functionality of the program is to seek the position of a value x in a global variable array A. If the value is not yet stored in the array, it is appended as a new value at the end

of the array. Furthermore, the program maintains the length of the A-array as a global variable m. Furthermore it maintains for each value in the A-array the number of times the value has been searched for. Another global array B is used for this purpose, i.e., for each index i the value stored in $B[i]$ equals to the number of searches for the value stored in $A[i]$.

Listing 6.6 'go to'-Program for seeking the position of a value in an array according to [204].

```
for i:=1 step 1 until m do
  if A[i]=x then go to found fi;
not found: i:=m+1; m:=i;
  A[i]:=x;B[i]:=0;
found: B[i]:=B[i]+1;
```

The program in Fig. 6.6 is not a structured program. Listing 6.8 shows a structured program, which is an alternative implementation of the program in Listing 6.6. The program in Listing 6.8 is also taken from [204]. Knuth compares the programs in Listing 6.8 and Listing 6.6 in order to argue about structured programming. The formulation of the program in Listing 6.8 slightly differs from the original presentation of the program text in [204] with respect to some minor changes in the layout. For example, we have put each statement on a different line and have given line numbers to the lines. Similarly, we have reformulated the program in Listing 6.6 resulting in the program in Listing 6.7. Despite the changes to the program layout, which can be neglected, the version of the program in Listing 6.7 uses a while-loop instead of a deterministic for-loop and therefore needs an incrementation of the loop variable i in the loop body. However, these changes make no difference to the discussion conducted in the following. We have used a formulation with a while-loop to make the program more directly comparable to the program alternative in Listing 6.8.

Listing 6.7 Reformulation of the 'go to'-Program in Listing 6.6

```
01 i:=1;
02 WHILE i<=m DO BEGIN
03   IF A[i]=x THEN GOTO 10
04   i:=i+1;
05 END;
07 m:=i;
08 A[i]:=x;
09 B[i]:=0;
10 B[i]:=B[i]+1;
```

The program in Listing 6.7 is realized with a while-loop that steps through the values in array A. In the loop the program compares the current value in the array with x and if the value equals x the while-loop is exited with a 'go to'-statement. The while-loop is exited with a 'go to' because the loop variable i holds the index of the first field that equals x, which is then exploited in the update of the corresponding field in the B-array in line '10'. Furthermore, the 'go to' is needed to circumvent the execution of lines '07' through '09' in case the value x actually exists in array A, because these lines are only there for handling the case when the value has not been found.

If x exists in A, the program in Listing 6.7 actually finds the first position of x in the A-array. However, only under the assumption that the following pre-condition and side-condition hold the values occur at unique positions in the array, i.e., a value's first position in the array is a value's single position. The necessary side-condition is that the piece of code in Listing 6.7 is the only means to update the A-array and the variable m. The necessary pre-condition is that the variable m holds the length of the array A, in particular, this means that the variable m must be set to zero and the array A is considered as empty before the first run of the program in Listing 6.7. The arrays A and B together can also be interpreted as a hash map data structure with array A holding the keys and array B holding the values. In that sense the program has the functionality of incrementing a key's value if applied to an existing key x, or inserting a key and initializing its value to 1 if applied to a key not yet existing in the hash map.

Listing 6.8 is a reformulation of the program in Listing 6.7 as a structured program, i.e., an alternative implementation without a 'go to'-statement. The condition that leads to an early exit from the loop in Listing 6.7, i.e., an exit before the loop condition has become false, has now been made part – in its negated form – of the loop condition in Listing 6.8. Then, after the loop, it is tested whether the loop has not been ended early by testing for the negation of the loop condition from Listing 6.7. The test guarantees that the code lines that are reserved for cases in which value x does not exist in array A are actually not executed whenever the loop has been exited early.

In [204] Donald E. Knuth argues that the program in Listing 6.7 is an example of a functionality for which structured programming is inadequate. In [204] Knuth gives two reasons for the asserted inadequacy. The first reason is the argument that the program in Listing 6.8 is slightly slower, because of the extra test of the loop condition after the loop in line '05'. Let us assume that the program is not part of some absolutely time critical application, i.e., we do not argue here at the level of machine programming but rather at the level of application programming. Then we can neglect this first aspect against the background of today's computing power. We think that for the most application domains the overhead of this single statement can be considered as marginal. Actually, we do not precisely know which of the programs is faster, because it could be, for example, that the realization of the 'go to'-mechanism

Listing 6.8 Structured Program for seeking the position of a value in an array according to [204].

```
01 i:=1;
02 WHILE (i<=m and (NOT (A[i]=x))) DO BEGIN
03   i:=i+1;
04 END;
05 IF NOT (i<=m) THEN BEGIN
06   m:=i;
07   A[i]:=x;
08   B[i]:=0;
09 END;
10 B[i]:=B[i]+1;
```

of the concretely used compiler and run-time environment is inefficient so that it avoidance outweighs the drawback of the extra test.

The second reason given by Knuth in [204] why he believes that the program in Listing 6.8 is less adequate than the program in Listing 6.7 is much more interesting for our discussion here. The reason given is Knuth's opinion that the program of Listing 6.8 is less readable than the program in Listing 6.7. The problem with this argument is that the better readability of Listing 6.7 is in our opinion not evident. We delve into a discussion of readability of program texts in general and the readability of the program texts in the Listings 6.6 through 6.8 now in Sects. 6.3.2 and 6.3.3.

6.3.2 Readability of Program Texts

Readability is a concept that is hard to grasp. Readability is a concept that inherently is about the perception of a person, i.e., about a person's disposition. A concept close to readability is the concept of understandability. It is possible to define some measure for the understandability of a program text. The question whether such a defined measure is actually objective is another question. One could instruct the test taker to read the program text and say 'stop' immediately when he thinks that he has understood the meaning of the program. Then, it must be determined whether the person has actually understood the program. If he has actually understood the program, the test is valid, otherwise it is not. The defined measurement is not feasible to compare the understandability of a program by a single person. Once, the person has read and understood the first version of the program, he has learned something and this will very likely impact the speed of understanding the second version of the program positively. But the measurement can be used at a statistical scale [13] by running the test with several groups to collect the average understanding periods for the several investigated program versions. Still, the defined measure is flawed. How to judge fair whether a person has actually understood the program? If a person simple describes in natural language the

step-by-step operation of the algorithm this may not be sufficient in order to be convinced that the person has understood what the algorithm does. What if people who answer particularly quickly tend to not really understanding the program?

We think that it is hard to tell whether the program in Listing 6.7 is more readable or more understandable then the program in Listing 6.8 or vice versa. Listing 6.6 shows the original version of the program in Listing 6.7. In our version the line number '10' serves as a label of the statement in line '10' when it is referenced in the 'go to'-statement. In the original version the statement has an explicit label 'found:'. Furthermore, the statements in lines '07' through '09' in Listing 6.7 are placed together in a single line in the original version, which visually emphasizes that these statements logically belong to the block of logic which is executed in cases where the value x is not yet stored in the array A. The readability of a program text can be improved by several means, e.g., line indentation, proper comments, or telling names. Such actions are usually best if they adhere to a style guide defined for a project. We think it is hard to tell whether the explicit exit from the loop with a 'go to' in Listing 6.7 or the exit via the loop condition in Listing 6.8 is more understandable. Actually, we are somewhat biased in favor of the exit via the loop condition.

Further Attempts to Improve the Readability of a Program Text

Various attempts can be undertaken to improve the readability or understandability of the program in Listing 6.8 further. In the program in Listing 6.9 we have made the usage of the logic that is needed to handle the single cases of loop exit unique. In Listing 6.8 the statement in line '10' is used in the case that the value x occurred in array A to increase the number of times it has been searched for the value x by one. However, the statement is also used in the other case, i.e., when the value x has not occurred in the array. Therefore, the number of times of searches for x is initially set to zero in line '08' in those cases so that it can be increased correctly to one afterwards. We think that it is somehow artificial to set the value of search times to an incorrect value first. Therefore we made the blocks for handling the two cases under consideration unique in the sense that there is no overlap in case handling any more.

In Listing 6.9 the lines '06' through '08' are only used if x has not occurred in the array and line '11' is only used if x has occurred in the array. Furthermore, lines '06' through '08' in Listing 6.9 completely handle the case that x has not occurred in the array, because we changed line '08' so that it immediately sets the value of search the time to the finally correct value of one. We did something else that improves the readability further in our opinion. In lines '07' and '08', which assign values to fields of the arrays A and B we have used the variable m instead of the variable i to index the wanted field. We think that the usage of m indicates better that we currently access the last element of each field, which fits better to the containing block.

Listing 6.9 Making unique the finalizing actions that react on the single conditions of a composed loop condition.

```
01 i:=1;
02 WHILE i<=m and (NOT (A[i]=x)) DO BEGIN
03   i:=i+1;
04 END;
05 IF NOT (i<=m) THEN BEGIN
06   m:=i;
07   A[m]:=x;
08   B[m]:=1;
09 END ELSE BEGIN
10    B[i]:=B[i]+1;
11 END;
```

Listing 6.10 shows yet another program solution for the discussed functionality. Here we moved all the specific code that is needed to handle the two cases leading to the exit of the loop inside the loop. We have done that at the price of an auxiliary variable ‘stop’ which has the sole purpose to signalize that the loop should be exited. Both cases that lead to an exit of the loop are detected and handled completely inside the loop. In the program version in Listing 6.9 each condition is tested twice, once as part of the complex loop condition and once again after the loop has been exited. The complex condition in Listing 6.9 necessarily says something about the reasons why the loop is eventually exited, because it conducts the respective test. The loop condition in Listing 6.10 is merely about encoding a control flow issue, however, it is also possible to introduce a comment explaining why the loop stops after the statement in line ‘03’ or it is also possible to encode this comment in the naming of the loop condition variable, e.g., by using a name like

- ‘NotYetCompletelyScannedAndNotYetFound’ or
- ‘((i<=m) AND (NOT A[i]=x))’

instead of the straightforward name ‘stop’. It is up to the reader to decide whether the program design pattern used in Listing 6.10 is more counter-intuitive or more intuitive than the one used in, e.g., Listing 6.9 – we do not really have an opinion about that. The program in Listing 6.10 is important for another reason. It hints at a general solution to resolve program cycles that have more than one exit by the introduction of extra state and extra logic.

6.3.3 Structured Programming and Denotational Semantics

In Sect. 6.3.2 we have discussed the concept of readability of program texts. We want to talk about readability of programs further but from a more for-

Listing 6.10 Moving special actions that react on the single conditions of a composed loop condition into the loop.

```
01 stop:=false;
02 i:=0;
03 WHILE (NOT stop) BEGIN
04   i:=i+1;
05   IF i>m THEN BEGIN
06     m:=m+1;
07     A[m]:=x;
08     A[m]:=1;
09     stop:=TRUE;
10   END ELSE BEGIN
11     IF A[i]=x THEN BEGIN
12       B[i]:=B[i]+1;
13       stop:=true;
14     END;
15   END;
16 END;
```

mal viewpoint this time. We take forward the main argument of this section, which is vague or informal, but, nevertheless, is an argument. It is fair to consider programming languages with a standard denotational semantics and their programs as better understandable than programming languages with a continuation-based semantics. Imperative programming languages with a standard denotational semantics are those that are completely block-structured, i.e., those that do not allow for arbitrary jumps, whereas programming languages with 'go to'-statements must be treated with a continuation-based semantics.

The discipline of formal semantics of programming languages could encourage us to argue further in favor of the program in Listing 6.8 as better understandable as the program in Listing 6.7. The program in Listing 6.8 can be given a standard denotational semantics [328, 285], whereas the program in Listing 6.7 cannot be given a standard denotational semantics but only a continuation-based denotational semantics [329, 330].

Basically, there are three different approaches to the formal semantics of programming languages, i.e., operational semantics [211, 351], axiomatic semantics [161] and denotational semantics [328, 285] which is also called the Scott-Strachey-approach to the semantics of programming languages. We do not want to delve into a distinction of these three approaches to semantics of programming languages. We just give a short statement about the essence of each of the three approaches. An operational semantics describes the effect of a program as its interpretation by a machine or, more abstractly, through the application of a reduction system to it. An axiomatic semantics tries to characterize the impact of the single building blocks of a program onto the

program state logically by identifying pre- and post-conditions for them. A denotational semantics directly assigns a mathematical object to a program as its semantics. Even in the standard case of functional programming languages or imperative programming languages without jumps, these mathematical objects cannot be just functions between ordinary sets. The mathematical objects are more complex, because in general a semantics for such a language has to deal with recursion, non-termination and higher-order functions. Therefore the mathematical objects are complete partial orders – or lattices [317] in the original work on denotational semantics – and continuous functions. Again, we do not want to delve into a complex discussion of formal semantics here, in particular, we do not want to delve into the technical aspects of denotational semantics. What counts is an understanding that denotational semantics is a mature apparatus in understanding the programs of a programming language directly as mathematical objects. When we say that the quality of denotational semantics lays in its direct understanding we mean that the semantics of a program is given by semantic composition of the semantics of its direct parts.

A major aspect of the denotational semantics of a programming language is that it is decompositional. As we have said, the semantics of a program can be understood directly by the application of a semantic function on the semantics of its sub programs. The semantics of an imperative program is the transformation of a store. Given an arbitrary store, it transforms it into another store. A store is a mapping that maps variables to values. The transformation on stores that is specified by a program can be considered mathematically as a function between sets. The sets must be special sets, i.e., complete partial orders, and the function must be special functions, i.e., continuous function, so that the denotational semantics works, however, for us it is sufficient to understand here that the transformations are functions. Stores are also functions. The source domain of a store is the set of variables and the target domain is the set of possible values.

The difference between the programs in Listings 6.7 and 6.8 is the following. For the program in Listing 6.8 a semantics can be given as a transformation of stores for all of its sub programs and recursively all the sub programs of sub programs. This is not so for the program in Listing 6.7, because of the 'go to'-statement that jumps out of the loop. In order to deal with this the sub programs must be described relatively to an extra notion of continuation [329, 330], which binds semantics of exploited programs to labels used in the exploiting program. The continuation of a program is a second argument in addition to the store – it is a kind of environment. A continuation-based semantics transports the operational concept of jumps to the mathematical structures that denote programs and therefore brings the operational complexity of them to these structures. The current discussion can be considered as a reformulation of the discussion on decompositional semantics of block-structured versus arbitrary business process models that we have conducted in Sect. 6.2.2 and illustrated Fig. 6.6 for the field of programming languages,

$$[[01]] = \lambda\sigma.\lambda v. \begin{cases} 1 & , v = i \\ \sigma(v) & , else \end{cases} \tag{6.1}$$

$$[[03]] = \lambda\sigma.\lambda v. \begin{cases} \sigma(i) + 1 & , v = i \\ \sigma(v) & , else \end{cases} \tag{6.2}$$

$$[[02..04]] = \nu\lambda F.\lambda\sigma. \begin{cases} ([[03]] \circ F)\sigma & , \sigma(i) \leq \sigma(m) \wedge (\sigma(A))(i) \neq \sigma(x) \\ \sigma & , else \end{cases} \tag{6.3}$$

$$[[06]] = \lambda\sigma.\lambda v. \begin{cases} \sigma(i) & , v = m \\ \sigma(v) & , else \end{cases} \tag{6.4}$$

$$[[07]] = \lambda\sigma.\lambda v. \begin{cases} \lambda p. \begin{cases} \sigma(x) & , p = \sigma(i) \\ (\sigma(A))(p) & , else \end{cases} & , v = A \\ \sigma(v) & , else \end{cases} \tag{6.5}$$

$$[[08]] = \lambda\sigma.\lambda v. \begin{cases} \lambda p. \begin{cases} 0 & , p = \sigma(i) \\ (\sigma(B))(p) & , else \end{cases} & , v = B \\ \sigma(v) & , else \end{cases} \tag{6.6}$$

$$[[06 \cdots 08]] = [[06]] \circ [[07]] \circ [[08]] \tag{6.7}$$

$$[[05 \cdots 09]] = \lambda\sigma \begin{cases} [[06 \cdots 08]]\sigma & , \sigma(i) > \sigma(m) \\ \sigma & , else \end{cases} \tag{6.8}$$

$$[[10]] = \lambda\sigma.\lambda v. \begin{cases} \lambda p. \begin{cases} \sigma(B)(\sigma(i)) + 1 & , p = \sigma(i) \\ \sigma(B)(\sigma(p)) & , else \end{cases} & , v = B \\ \sigma(v) & , else \end{cases} \tag{6.9}$$

$$[[01 \cdots 10]] = [[01]] \circ [[02 \cdots 04]] \circ [[05 \cdots 09]] \circ [[10]] \tag{6.10}$$

however, this time against the background of a more formal treatment which is available for programming languages, i.e., denotational semantics.

As an illustration of what we have just explained we glimpse over the denotational semantics of the program in Listing 6.8 in a step-by-step fashion now. The denotational semantics of the program in Listing 6.8 is given by the Eqns. 6.1 through 6.10.

The statement in line ‘01’ assigns the value 1 to the variable i. This is also expressed by the denotation of the statement that we have given in Eqn. 6.1. The expression [[01]] is a shorthand notation for the semantics of the line ‘01’, in which we have used so-called semantics brackets [[and]]. The denotation of line ‘01’ is a function that takes a store σ – the input store – as an argument and yields a new result store. The input store and the result store are both functions that map each variable to a value. The result store maps each variable v to the value that it is mapped to by the input store σ except for

the variable i which is mapped to the value 1 by the result store independent of the value it is mapped to by the input store. This is exactly what the semantics of an assignment statement is about, i.e., manipulating the left-hand variable and keeping all other variables of the store as they are.

The semantics of line '03' is given in Eqn. 6.2. Line '03' is similar to line '01'. Here, the variable that is manipulated is again the variable i, however, this time it is increased by one, which means that the result store maps the variable i to the value that it is mapped to in the input store σ plus 1. The semantics of the while-loop in lines '02' through '04' is given in Eqn. 6.3. The semantics of $[[02 \cdots 04]]$ is defined recursively. It is defined as a function F that takes a store σ as an input argument. The function first evaluates the loop condition with respect to the input store. If the loop condition evaluates to false, the function yields the input store as it is as the result store, which represents adequately the termination of the while-loop. In the case that the loop condition evaluates to true, the function F is unfolded one time by applying the semantics of the inner block of the while-loop, i.e., $[[03]]$ in our case, to the input store, taking the result of this application of $[[03]]$ and then applying F recursively to this result. You might want to use the explicit notation $F\big([[03]](\sigma)\big)$ for this sequenced application of first $[[03]]$ and than F, however, we have decided to express it by a function concatenation $\big([[03]] \circ F\big)$ which is then applied to the input store σ as a whole.

You might also want to use a more direct notation for recursive definition like $\big(F \equiv_{DEF} G(F)\big)$ instead of the $\nu\lambda$-notation $\big(\nu\lambda F.G(F)\big)$ that we have used in Eqn. 6.3. However, the $\nu\lambda$-notation expresses better that recursive definitions have no operational semantics but the so-called fixed point semantics in the denotational approach to the semantics of programming languages. Without further elaboration and explanation the understanding of a recursive definition is operational, i.e., it relies on a notion of reduction of the program text which is reduced until a token is reached that leads to the next reduction of the entire program text. The fixed point semantics is another viewpoint on recursion. It defines the recursive function $\big(F \equiv_{DEF} G(F)\big)$ as the smallest fixed point of the higher oder function $\big(\lambda F.G(F)\big)$. At a first sight, the fixed point semantics is less operational, however, it is also somehow operational by the way the smallest fixed point is constructed as the limit of the endlessly repeated application of the considered higher order function to the bottom element $\perp$ of its input domain – see the fixed point theorem of Knaster-Tarski [336] and, e.g., [145] for further reference.

We have said that the stores that are transformed by programs are mappings that map variables to values. In our case the store is nested for some variables, i.e., the array variables. First the application of the store to an array name yields a further function that maps position indexes to result values. Then, the application of this function to a concrete position index yields the array value. For example, the expression $(\sigma(A))(i)$ in Eqn. 6.3 stands for the concrete array value $A[i]$. According to this, the updates to array values are

modeled in Eqns. 6.5, 6.6, and 6.9. The semantics of the program lines '06' through '08' should be self-explaining now.

The semantics of a sequence of program statements is given by the concatenation of the function that they denote. The semantics of the program that consists of the lines '06' through '08' in Eqn. 6.7 is an example for this. The semantics of the lines '05' through '09' is now defined by Eqn. 6.8. The lines '05' through '09' form a case construct. Equation 6.8 checks the case condition on the store. If it evaluates to false the result store remains the same, otherwise the result store is yielded by the application of the semantics of the lines '06' through '08' to the store. The semantics of the program line '10' in Eqn. 6.9 should be again self-explaining. Finally, the semantics of the whole program is defined as the concatenation of the semantics of its direct program parts in Eqn 6.10.

In the example we have constructed the program semantics bottom up. The example has shown that program semantics can be given by composition of the semantics of its program parts. In the case of programs without 'go to'-statements the semantics of all program parts can be given homogenously as transformation of stores. This is not so for programs with 'go to'-statements. This means that this is not so for programs with 'go to'-statements in general. Some programs with 'go to'-statements can also be given a standard semantics. We call a semantics that is given homogenously as transformation of stores for all program parts a standard denotational semantics or standard semantics for short. If a program with 'go to'-statements has only such circles that can be exited by no more than one means, it can also be given a standard semantics. The program in Listing 6.7 has no standard semantics, because the circle realized by the program while-loop in lines '02' through '05' is a circle that can be exited by two means, i.e., the loop-condition in line '02' and the 'go to'-statement in line '03'. Therefore, the program in Listing 6.7 must be interpreted as a program of a programming language with 'go to'-statements and, as we have said earlier, a programming language with 'go to'-statements does not have standard denotational semantics but must be given a non-standard denotational semantics, e.g., a continuation-based denotational semantics.

6.4 Frontiers of Structured Business Process Modeling

We are not able to characterize the subset of business processes for which we believe that structured process descriptions, i.e., structured business process models, are better. However, we have argued against the hypothesis that this subset equals the set of all business processes. This means, we say that the postulation that all business processes should be structured [162] is arguable. We do not say that the statement is false; this would be to harsh, because the message that all business processes should be structured is not a statement with a truth value but a postulation. Furthermore, we argue in a yet informal

setting, which is the pragmatics of system specification methodology. For the same reason, we would say that it is not appropriate to say that the message is true.

However, eventually we dare to say: It should not be postulated that all business processes should be structured. This is a negative result. Nevertheless, it is an important result, because it can help modelers in preventing pitfalls.

Structuring is considered a proven concept in program design. We have discussed structured programming in Sect. 6.3. The overall question behind the discussion in Sect. 6.3 is whether structured programming is actually as proven as it is considered. We have discussed an example program of the established computer science author Donald E. Knuth that he actually considers as a counter-example for structured programming. What interests us here is merely the fact that an established computer science author argues against the general validity of the structured programming paradigm. Actually, Knuth has argued that the non-structured program in Listing 6.7 is somehow better than the structured program version in Listing 6.8, in particular also because, in his opinion, it is slightly better readable than the program version in Listing 6.8. We have said that we do not think so and that we actually feel that the structured program version in Listing 6.8 is better readable. In that sense we have actually defended structured programming. For us, it is important to show, that the standing of structured programming should not be used without care in the argumentation for a structured approach to business process modeling.

The arguments that are given in favor of structured programming:

- Improved readability.
- Improved maintainability.
- Improved testability.

With respect to validation the arguments have different levels of formality. The longer the structured approach is used as the best practice in real-world projects, the more we rely as a software engineering community in the validity of its claims. Empirical software engineering [36] with its experiments could offer tools for the more systematic evaluations of a software engineering approach, pattern, method and so on. The arguments in favor of a structured approach must be basically the same for the domain of programming languages and the domain of business process modeling language. And the same holds for the claims of structured business process modeling as holds for the claims of structured programming. Without systematic investigation they must be proven through experience in real-world projects and feedback from software engineers in real-world projects.

However, what we have tried to argue in this section is that the standing of structured programming can not simply be transferred to a structured business process modeling approach, even if it would be taken for granted that structured programming is without doubt the best practice in program

design. We summarize the reasons for this in the following. The semantics of programming languages is different from the semantics of business process modeling languages. The semantics of a program is a state transformation. The state transformation of a program is achieved compositionally by the parts of the program which again have state transformations as semantics. It does not matter, how a program achieves the overall state transformation with its inner state transformations. This fact opens a space for program design. A program with better design, i.e., with improved readability, maintainability, testability and the like can be taken to replace a program with a weaker design. Business processes may also manipulate state and therefore may cause a state transformation as a side-effect. However, they have an observational semantics. In particular, two business process descriptions are considered as equal if they are observational equivalent. For a program, the processing does not count – as long as we have not to deal with reactive programming and the like. What counts for a program is the result. Two programs are equal if they evaluate to the same result; they are equal if they stand for the same state transformation.

Two business process descriptions may not be interchangeable even if they always have the same effect with respect to a business objective. This is because each modeling element stands for a real-world activity and a business process is a plan for doing work. Therefore, the opportunities for reshaping a given business process description are in general much more restricted than the possibilities for reshaping a program. You can reshape a program as much as you want and, in particular, as much as you think it improves the design of the program text, as long as you ensure that the resulting reshaped program has the same semantics. It is even a best practice to abstract from the implementation, to hide the implementation of functionality. All this does not immediately hold also for business process modeling. We give a further small example for the limitation in freedom for reshaping in Fig. 6.14 that works even without a discussion of structuring principles and structured modeling languages.

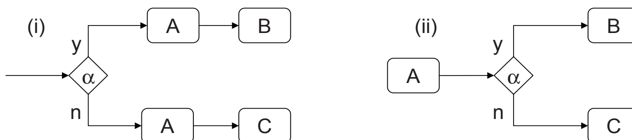

Fig. 6.14. Two business processes that are not behavioral equivalent.

Figure 6.14 shows two processes. In process algebraic notation [249, 250] process (i) is $\alpha.A.B + \neg\alpha.A.C$ and process (ii) is $A.(\alpha.B + \neg\alpha.C)$. Now, let us assume that A has no impact on the outcome of the decision α. If we interpret the two processes (i) and (ii) as programs of a usual programming language with A, B and C being statements, this would mean that A is not

manipulating those parts of the system state that is tested by α. If (i) and (ii) were programs we now could start a discussion on whether we would prefer the program design of (i) over the program design of (ii). For example, the inner reuse of activity A in (ii) is a plus, because it improves maintainability. But (i) and (ii) are business process description and the discussion about which one has the better design is limited even if α is independent of A and, therefore, A and B have the same effect with respect to the business objective. The problem is that also α describes an activity, i.e., a test that is conducted, that occurs in the real world. And it might be a huge difference whether you conduct the test at the beginning of the process or conduct it in the middle after activity A for several reasons. One such reason might be that α tests some property of a good that flow through A and the good occurs at another location after A than before A and the location of the good before A is much more suited for conducting the α-test than the location after the processing of A.

The described problem of reshaping a business process description – against the fact that business process modeling must be in first place domain-oriented and only in second place artifact-oriented – is the basic motive also in the other examples of this section.