Deriving Classes from Scenarios in Object-oriented Design

(Preliminary version)

Kurt Nørmark
Aalborg University
Denmark*

May 2, 1997

Abstract

Scenarios can be used to model the dynamic aspects in an object-oriented design process. This is attractive because scenarios allow the designer to specify the way objects interact at a tangible and concrete level of abstraction. Although scenarios are based on examples, the scenarios represent a holistic view on the object system in contrast to the fragmented and decentralized specifications in the individual classes. This paper deals with the problem of extracting static model information (about classes and methods) from a dynamic model (objects and scenarios in term of message hierarchies). The paper is based on the dynamic modelling language from DYNAMO, and supported by the set of DYNAMO tools.

1 Introduction

In the object-oriented design process we make a variety of different *models* of programs. Some of these models are oriented towards the *static structures* of the design whereas others are oriented towards the *dynamic structures*. Models of static structures include class diagrams, which show a number of relations among classes, methods, and attributes. Models of dynamic structures focus on objects, relations among objects, and interactions among objects.

In the work, on which this paper is based, we focus primarily on models of the dynamic structures. The reason is that we hypothesize that program designers primarily think in terms of objects, object relations, and object interactions during the creative phases of the design process. If this is true, it is not optimal to express the design at a static level. This holds in particular in situations where the designer handles complicated object interplays. Rather, the designer should be able to express himself or herself in terms of dynamic structures early in the design process. In that way, it becomes possible to shape the design at a concrete and tangible level already at the outset of the design efforts, namely in terms of objects, object relations, and object interactions.

*Department of Computer Science, Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark. E-mail: normark@cs.auc.dk. WWW: http://www.cs.auc.dk/~normark/ — This research was supported by the Danish Natural Science Research Council, grant no. 9400911.

Although it is our hypothesis that models of dynamic structures are easier and more natural to deal with in certain design situations, it is, of course, attractive and necessary to elaborate models of the static structures as well. Such 'static models' are more abstract than the 'dynamic models' because a number of properites of objects and messages are concentrated to shared properties of a single class or method. Furthermore, a static model is closer to the source program, which must be written sooner or later in the development process, and which, after all, is the most central model at all in the entire develop efforts.

In this paper we will discuss how to generate automatically some elements of the static models from the dynamic models. More concrete, we will discuss to which degree it is possible to generate classes and methods from scenarios. In our context, a scenario is an example of object interactions, typically crossing the abstraction barriers defined by the underlying classes.

Figure 1 shows an informal descrition of a scenario. This particular scenario is related to the command design pattern [2]. The idea behind the command design pattern is to turn interactive commands (such as a 'cut' and 'paste' editing commands) into objects, and to install such command objects in invoker objects (such as in pop menu items). When the command is installed, it receives information about a receiver object (such as the document being edited), which will be affected by executing a command.

```
* Request command execution in a client object.
  * This leads to a request about command execution in the
    invoker object.
    * In turn, this leads to execution of the command object,
      which is associated with the invoker.
        * The command object stores undo information in itself,
          hereby preparing a subsequent undo operation.
        * The command object executes itself by changing
          the state of the receiver object.
```

Figure 1: *An informal description of a scenario.*

Because of the duality between classes and objects, methods and messages, and object relations and class relations, it is should not come as a surprise that certain static OOD structures can be generated from dynamic structures, as present in a set of scenarios.

In the extreme, one could propose automatic generation of an object-oriented source program from examples. We do not, however, believe that this is an effective way of producing high quality programs. But we do believe that an automatic synthesis of classes are valuable *summaries* of a (possibly) great number of scenarios, which cover a number of essential examples of object interplays. Furthermore, we think that the automatically generated classes, on the assumption of some basic forward and reverse engineering properties (discussed further in section 7) can be used as a skelleton in a refinement process on the way towards the final source program.

We do not focus on a particular programming language when we have to present the result of the class derivation. Instead, we support a range of different object-oriented programming languages, which can be selected as the target of the class and method derivations. It should be noticed that the selection of programming language is more than just different ways of pretty printing a number of fixed constituents. The inherent differences among the languages affect the task of expressing the derived classes and methods.

In this paper we will first describe the scenario concept. This is the foundation of the rest of the paper, because the derived classes and methods depend exclusively on the information which is present in the scenarios. In section 3 we explain in details how we derive methods from one

or more messages in a set of scenarios. In section 4 we continue with the derivation of classes. Prior to the conclusions and the discussion of similar work we briefly outline the tool support for method and class derivation in DYNAMO.

2 The scenario concept.

The scenario concept of DYNAMO is simple and straightforward compared to similar concepts from other work, such as the SCED system [4]. The main reason is that we keep the scenarios clean from control mechanisms such as selection and iterations. One of the main points in this paper will be to demonstrate how method outlines (containing control structures for selection and/or iteration) can be derived from a set of simple scenarios.

A *scenario* is described in terms of a message from the "surrounding" to a receiver object R. The receiver, may in turn, send a number of messages to other objects (including R), and so on recursively. Thus, a scenario can be thought of as a tree of messages.

A *message* M is characterized by:

- A receiver object, which we call *receiver*(M).
- The actual parameters of M.
- The situation *pre-situation(M)* which occurs just after the parameters have been passed (see below).
- A general understanding, which explains M in general.
- A specific understanding of the particular example of the message.
- The objects provided by M (see below).
- The messages sent from *receiver(M)* to other objects.
- The result of the message, when sent to *receiver(M)*.
- The situation *post-situation(M)* which occurs just before M is completed, and is about to gain control to the sender.

The pre-situation and post-situation are assertions, which describe the situations at some given points of time during a scenario. The assertions, which in the current version of DYNAMO are supported as an informal element, are *situational* in contrast to *prerequisitional*. This means that a pre-situation or a post-situation assertion describe the situation per se, at the point where it is located. A prerequisitional assertion expresses a requirement to the state of the computation. As an example, prerequisitional assertions are used in the programming language Eiffel [5] as pre-conditions of operations.

Objects enter the scene (the set of objects which exist at a given point in time) via a mechanism called *object provision*. Relative to a message, an object may be provided in the parameter list, in its list of object provisions, and as part of the result. An object provision is a convenient mechanism which states the relevance of an object exactly at the time it is needed by the designer. In some cases a provided object may actually have existed for a some time, but in the current scenario it first becomes relevant at 'object provision time'. In other situations it may be the intention of the designer to actually create the object at the provision time. In these situations, object provision is identical to object creation. Anyway, an object provision claims

```
Send message REQUEST-COMMAND-EXECUTION() to A-CLIENT.
  Send message REQUEST-COMMAND-EXECUTION() to INV.
    Provide CMD [ass from INV] as cmd is the object which,
                via setup, has been associated from the invoker
    Send message EXECUTE() to CMD.
      Provide REC [ass from CMD] as the object on which the command
                  ultimately is carried out
      Send message SAVE-UNDO-INFORMATION() to CMD.
      RESULT: undo information has been saved
      Send message DO-ACTIONS() to REC.
      RESULT: command has been effected
    RESULT: the effect of command has taken place
    Send message ADD(CMD ) to COMMAND-HISTORY.
    RESULT: the parameter has been added to the list
  RESULT: The command has been executed
RESULT: Done
```

Figure 2: *A trace of the informal scenario from figure 1.*

the existence of an object which possess certain relationships to the already existing objects on the scene. As a simple and practical convention, all objects on the scene have a unique name through which they can be referred to during a scenario. Other kinds of object handles (such as dot notation and references) are considered irrelevant for design purposes. In addition, the class of an object is also registered.

The *understandings* are informal descriptions, in pure text. The understandings are crucial in order to freeze and maintain the designer's intuition about the message.

The *result* of a message is an informal description of the effect in case the message activates a procedural abstraction. If the message activates a functional abstraction, the result may be an existing object being returned, a provision of an object to be returned, or a non class-bassed value.

A scenario can be used to describe object interaction across the usual abstraction barriers. These abstraction barriers are carefully protected in the static, class-based models as well as in the implementation phase of the program development. However, in the early design phase it is often useful to describe the interaction across these barriers, because such interaction may give a better picture of the way we intend to solve a problem.

Each example of a message m to some C-object (some object of class C) will in the sequel be called a *message case for m in C*. When talking about message cases we do not care whether the message is sent at the outer level of a scenario, or some place deeper in a scenario.

A more complete and thorough description of scenarios, messages, objects, scenes, object-provisions and other DYNAMO concepts can be found in [6].

Figure 2 shows an example of a scenario (the one described informally in section 1). We here present the scenario as a *trace* of the most important information in the scenario. A trace is roughly a pre-order traversal of the tree of messages in the scenario. The trace is generated by the DYNAMO environment.

In section A.1 of the appendix we give an abstract grammar of a dynamic model. The stuff in the appendix therefore complements the description above.

3 Deriving methods.

In this section we will discuss how to derive information about a method in a class on the basis of a number of scenarios.

3.1 Basic method derivation.

Let us assume that we are interested in generating an outline of the method M in the class C. In the simple case, the set of scenarios, on which we are based, only contain a single message case *mc* for m in C. I.e., the entire set of scenarios only contain one example of an m-message to a C-object. Consequently, the method needs to be generated from a single example.

There is a number of elementary informations in *mc*, which can be used when generating the outline of the method, which we here call M:

- The *name of the message* can be used as the *name of the method*.
- The *actual parameters* can be used to conjecture some similar *formal parameters* together with the classes of these.
- The *general message understanding* is suitable as the *method comment*. We also support a specific message understanding, which intented to explain the particular message case. This information is not appropriate here.
- Based on the *result of the message* it may be derived whether the method is *procedure-like* or *function-like*. In some programming language this gives rise to different renderings of the method. In case of a function-like method, we can extract information about the type of the object which will be returned from the method. This is relevant for programming languages with static typing.
- The *pre-situation and post-situation assertions* of a message may, in some circumstances, be turned into method *pre-conditions and post-conditions*, respectively. However, the differences between situational and prerequisitional assertions (cf. the discussion in section 2) must be taken into consideration when doing so. Depending the programming language in which we render the method, this may be a formal element, or just part of a method comment.

In case there are local object provisions in *mc*, these can be translated to declarations of local variables of the method M, which refers to appropriate objects. Thus, object provisions of local objects leads both to declaration of local variables and to initialization of these variables to objects. The latter aspect is, in general, the most problematic with respect to 'code generation'.

In the most elementary situation, the message case does not prescribe messages to other objects on the scene. We say that the message is *terminal*. This is either because no additional communication is necessary in order to fullfil the obligations of the underlying method, or because the additional communication from *receiver*(m) to other objects is irrelevant for the model we are building. Needless to say, it is not possible to generate a method body based on a terminal message case.

Let us now consider the case where we have a single, non-terminal message case *mc* for m in C. Thus, m sends messages from *receiver*(m) to a number of other objects (locally provided objects, parameters, or globally available obejcts) and/or to itself.

```
Request-Command-Execution IS
--The invoker wants to execute its command
DO
  Cmd.execute;
  Command-History.add(Cmd);
ENSURE The command has been executed
END;
```

Figure 3: *Derivation of a method from a non-terminal message based on a single message case.*

Figure 3 shows a simple example of deriving a method from the message `request-command-execution` to an invoker (the inner-most `request-command-execution` message in figure 2). This messages happens to represent the only message case of the message `request-command-execution` in the class `Invoker` in this particular dynamic model. As it appears from the figure we show the message in an Eiffel-like syntax. The derivation of the method is taken directly from the DYNAMO environment.

3.2 Conditional method derivation.

Let us now assume that there are two or more message cases that contribute to the method we are deriving from the scenarios. The challenge is here how to subsume a number of examples (found at various places in the scenarios) into a single description (a method body), which in some sense covers all the examples.

To be concrete, we will assume that there are n message cases mc_1, mc_2, ..., mc_n of the message m in a class C. Each message case involves a message m to some instance o_i of C. Message case mc_i, in turn, sends the messages

$$m_{i,1},\ m_{i,2},\ m_{i,n_i}$$

from o_i to the objects $o_{i,j}$, $i = 1...n$, $j = 1...n_i$. A concrete example, for three message cases is shown in figure 4. [1]

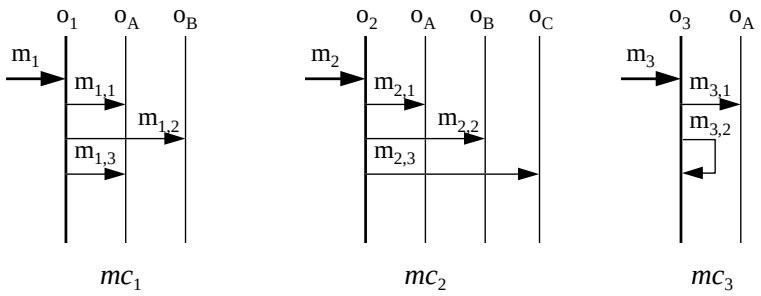

Figure 4: *Three message cases for m in a class C.*

For our purposes in this section, the pre-situation and the post-situation of a message play important roles. Recall that each message m, together with each message sent from *receiver*(m)

[1] In the figure, the following equations relate the notation in the figure to the general notation: $m_1 = m_2 = m_3 = m$, $n_1 = n_2 = 3$, $n_3 = 2$, $o_{1,1} = o_{2,1} = o_{3,1} = o_A$, $O_{1,2} = o_{2,2} = o_B$, and $o_{2,3} = o_C$.

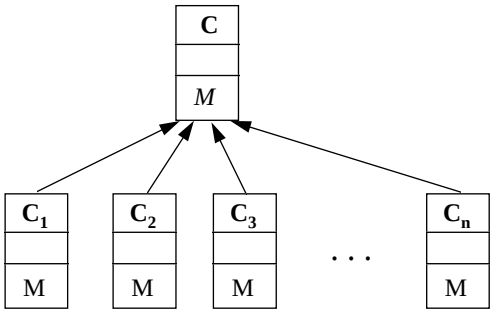

Figure 5: *A number of methods arranged in an inheritance hierarchy.*

to other objects, have pre-situations and post-situations. Thus, the message case mc_i gives rise to the following "Hoare statements" for $i = 1, ...n$:[2]

- $\{P_i\}$ o_i.m $\{Q_i\}$
- $\{P_{i,1}\}$ $o_{i,1}.m_{i,1}$ $\{P_{i,2}\}$ $o_{i,2}.m_{i,2}$ ... $\{P_{i,n_i}\}$ $o_{i,n_i}.m_{i,n_i}$ $\{P_{i,n_i+1}\}$

Here P_i = *pre-situtation(m_i)*, Q_i = *post-situation(m_i)*, $P_{i,j}$ = *pre-situation($m_{i,j}$)*, and $P_{i,j+1}$ is implied by *post-situation($m_{i,j}$)*, $i = 1...n$, $j = 1...n_i$.

It may be the case that the classes of the receivers of m_i, $i = 1..n$, are subclasses of C and that C contains a virtual method which "covers" all the message cases. This is illustrated in figure 5. In that situation, a specialized method is selected based on the class of the receiver object. As a consequence of this, it seems best to generate a number of different methods (up to n different methods) in subclasses of C. But as already discussed, we do not synthesize class hierarchies in our current work. Therefore we will in the following discussion assume that the n different message cases should be merged into a single method in C.

As a rough beginning, the method body of M in C is a n-way selection of one of the message cases. Given the three message cases from figure 4, the body of the derived method M goes like

[1] **if** *pre-situation(m_1)*
then $o_A.m_{1,1}$; $o_B.m_{1,2}$; $o_A.m_{1,3}$
else if *pre-situation(m_2)*
then $o_A.m_{2,1}$; $o_B.m_{2,2}$; $o_C.m_{2,3}$
else if *pre-situation(m_3)*
then $o_A.m_{3,1}$; self.$m_{3,2}$

Notice that we capture and reflect the situations where a message is sent to the "current object". In DYNAMO all objects are named. In case a message m_1 is sent to an object a, which in turn sends a sub-message m_2 to a, the latter message is rendred as "self.m_2" (all depending of the concrete syntax of the renderng, of course).

The n-way selection above is not always the outcome we want from the method synthesis. In some situations the message cases are variations of each other. One of the cases may reflect a normal case, the remaining may be abnormal cases. As an example, all the cases may share a common prefix but differ mutually in the suffixes.

[2] In the Hoare statements we ignore the actual parameters of the messages.

In order to illustrate this we can assume that $m_{1,1} = m_{2,1} = m_{3,1}$ in figure 4. In this context, two messages are equal if they have the same name and if their receiver objects are identical.[3] Let us call this message $m_{*,1}$. Thus, in any case we send the message $m_{*,1}$ before sending any other message in M. Consequently, a more appropriate method derivation looks like:

[2] $o_A.m_{*,1}$;
if *pre-situation($m_{1,2}$)*
then $o_B.m_{1,2}$; $o_A.m_{1,3}$
else if *pre-situation($m_{2,2}$)*
then $o_B.m_{2,2}$; $o_C.m_{2,3}$
else if *pre-situation($m_{3,2}$)*
then self.$m_{3,2}$

Notice that the conditions in the if-then-else control structure have been changed from the pre-situations of the original message cases to the pre-situations of the second "sub-message" of each message case. These new conditions are far more realistic in an implementation of M. We may say that the pre-situations in the first derivation (*pre-situation(m_i)*, $i = 1, 2, 3$) are *oracular assertions* which predicts what happens in one of the sub-messages.

Let us now assume that all but one of $m_{1,2}$, ... $m_{n,2}$ are equal (in the same sense as above). We talk about a *semi-prefix* if a sequence of messages is a prefix of all but one of a set of scenarios. In terms of our example from figure 4, where $n = 3$, we will assume that $m_{1,2} = m_{2,2} = m_{*,2}$. A typical set of scenarios leading to this situation sends $m_{*,2}$ in the case that $m_{*,1}$ succeeds; $m_{3,2}$ is sent in case $m_{*,1}$ fails:[4]

[3] $o_A.m_{*,1}$;
if *post-situation($m_{*,1}$)*
then $o_B.m_{*,2}$;
if *pre-situation($m_{1,3}$)*
then $o_A.m_{1,3}$
else $o_C.m_{2,3}$
else self.$m_{3,2}$

Notice that we take success and failure from the post-situation of $m_{*,1}$ rather than from the pre-situations of $m_{*,2}$. In practical situations this does not lead to any difficulties, because the designer is not likely to specify both a pre-situation of $m_{i,2}$ and a post-situation of $m_{i,1}$.

As an example we will study a set of scenarios of a simple programming environment. In this environment, there are three different scenarios of compiling a source program:

1. Complete compilation.
2. Non-complete compilation because of syntax errors.
3. Non-complete compilation because of type errors.

The traces of the three scenarios are shown in figure 6. The first of these scenarios represent the normal case, and the two others are abnormal. Notice that the two abnormal cases share

[3] We do, in addition, expect the messages to have similar or congruent parameter lists. However, this is an assumption which is not part of this kind of message-equality.

[4] In a more conservative program generation we would substitute the two "else" clauses with "else if" clauses.

```
Send message COMPILE(OBJECT-PROVISION UC: USER-COMMUNICATION [local]) to SOURCE.
  Send message LEX-ANALYZE() to SOURCE.
  RESULT: Provide TS [floating] as a list of tokens, as produced by a lexical analysis.
  Send message PARSE() to TS.
  RESULT: Provide AST [floating] as an abstract syntax tree as proced by the parser.
  Send message CHECK() to AST.
  RESULT: No fatal errors found
  Send message GENERATE-CODE() to AST.
  RESULT: Provide CODE [floating] as an program  object which may be directly  executed
  Send message SAVE(OBJECT-PROVISION NAME: STRING [part of SOURCE]) to CODE.
  RESULT: Code has been saved in the database
  Send message WRITE(DONE) to UC.
  RESULT: Compilation done and code saved.
RESULT: CODE


Send message COMPILE(OBJECT-PROVISION UC: USER-COMMUNICATION [local]) to SOURCE.
  Send message LEX-ANALYZE() to SOURCE.
  RESULT: Provide TS [floating] as a list of tokens, as produced by a lexical analysis.
  Send message PARSE() to TS.
  RESULT: Provide ERROR-AST [floating] as an error AST which represents a program which
          could not be parsed
  Send message WRITE(PARSE-PROBLEMS) to UC.
  RESULT: Encountered a context free syntax error.
RESULT: Provide ERROR-CODE [floating] as an object which represents impossible code


Send message COMPILE(OBJECT-PROVISION UC: USER-COMMUNICATION [local]) to SOURCE.
  Send message LEX-ANALYZE() to SOURCE.
  RESULT: Provide TS [floating] as a list of tokens, as produced by a lexical analysis.
  Send message PARSE() to TS.
  RESULT: Provide AST [floating] as an abstract syntax tree as proced by the parser.
  Send message CHECK() to AST.
  RESULT: No fatal errors found
  Send message WRITE(CHECK-PROBLEM) to UC.
  RESULT: Encountered a context sensitive error.
RESULT: Provide ERROR-CODE [floating] as an object which represents impossible code
```

Figure 6: *Traces of three different compilation scenarios.*

```
Compile(Uc: User-Communication): Intermediate-Code IS
--Try to compile the SOURCE object to an executable representation
DO
  Ts := Current.lex-analyze;
  Ast := Ts.parse;
  IF parsing ok THEN
    Some-var := Ast.check;
    IF checking ok THEN
      Code := Ast.generate-code;
      Code.save(Name);
      Uc.write(Done);
    ELSEIF SOURCE cannot be checked successful THEN
      Uc.write(Check-Problem);
    END
  ELSEIF Source cannot be parsed THEN
    Uc.write(Parse-Problems);
  END
  Result := Code;
END;
```

Figure 7: *The method derived from the three senarios in figure 6.*

the message `PARSE` to `TS`. The method derived from the three messages is shown in figure 7, again rendered in an Eiffel-like syntax. The method in the figure is generated by the DYNAMO environment from the three scenarios.

The derivations discussed above are, of course, not universal. We capture common prefixes and common "semi prefixes". There may be many other patterns, e.g., common suffixes instead of common prefixes. Our approach may be characterized as purely syntactical.

From an algorithmic point of view we represent a method body as a *conditional regular expression.* The first such expression is formed from all the message cases of m in C. Subsequently we make a number of transformations on the expresssion. First we find out if there exists a common prefix of $m_{i,1}$, $m_{i,2}$, m_{i,n_i}, $i = 1...n$. In other words we find j such that $m_{1,1}$=$m_{2,1}$=...=$m_{n,1}$, ..., $m_{1,j}$=$m_{2,j}$=...=$m_{n,j}$. I $j > 0$ the prefix exists, and we factor the prefix out of the selection. If $j = 0$ we look for an element in {$m_{1,1}$,$m_{2,1}$, ..., $m_{n,1}$} which only occurs once. If we assume that $m_{j,i}$ is such an element we look for a common prefix in {$m_{1,1}$,$m_{j-1,1}$, $m_{j+1,1}$ ..., $m_{n,1}$}. More algorithmic details can be found in appendix A.2.

3.3 Iterational method derivation.

As we have seen in the previous section, selections (if-then-else's) are obtained from multiple and different message cases extracted from a set of scenarios. In this section we will discuss how to derive methods with iterations.

The basic idea in this part of the work is to conjecture elements of iteration based on repetition in a single scenario. As a concrete example we will first study scenario 1 in figure 8.

Let us assume that

- $m_{1,2} = m_{1,4}$
- $m_{1,3} = m_{1,5}$
- *pre-situation*($m_{1,2}$) = *pre-situation*($m_{1,4}$)

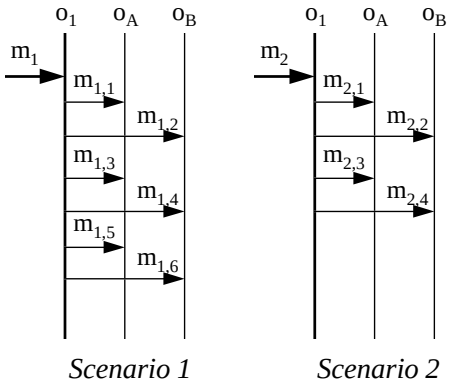

Figure 8: *Two scenarios with repetitions.*

In this context, two messages are equal if their message names are equal, if the receiver objects have the same name, and if the parameter lists are equal. Two parameter lists are equal if they are the same lenghts, and if the parameters are equal pairwise (comparing actual parameter names and classes in case we compare two object-provision parameters). Notice that the equality applied here is stronger than the equality applied in section 3.2 with the purpose of finding common prefixes.

Given scenario 1, and the assumptions from above, we will conjecture that the sequence $o_B.m_{1,2}$; $o_A.m_{1,3}$ is repeated. The rationale behind this conjection is that the situation P before each repetion is the same; thus it is tempting propose a generalization of scenario 1 which embed $o_B.m_{1,2}$; $o_A.m_{1,3}$ into a loop, which runs while P is true. The method derived by DYNAMO from scenario 1 in figure 8 is the following:

```
[4]  oA.m1,1;
     while pre-situation(m1,2)
     do begin
          oB.m1,2 ;
          oA.m1,3
        end;
     oB.m1,6
```

The conjecture of the loop above is based on the observation of a *complete repetiton*. In DYNAMO we also support conjectures of loop based on *incomplete repetitions*, but only at the *end* of a scenario. Scenario 2 in figure 8 is an example, in which we assume that $m_{2,2} = m_{2,4}$ in the same sense as above. Furthermore, we will assume that *pre-situation($m_{2,2}$)* and *pre-situation($m_{2,4}$)* are the samme. This leads us to conjecture that $m_{2,4}$ is the start of a repetion of $m_{2,2}$ and $m_{2,3}$. This facility is in particular useful if the repeated sequence is lengthy, because we hereby avoid duplication of many messages. Below we show the body of the method conjectured from scenario 2 of figure 8.

[5] $o_A.m_{2,1}$;
while *pre-situation($m_{2,2}$)*
do begin
$o_B.m_{2,2}$;
$o_A.m_{2,3}$
end

Let us finally discuss scenario 3 in figure 9 which is more difficult with respect to loop conjecturing.

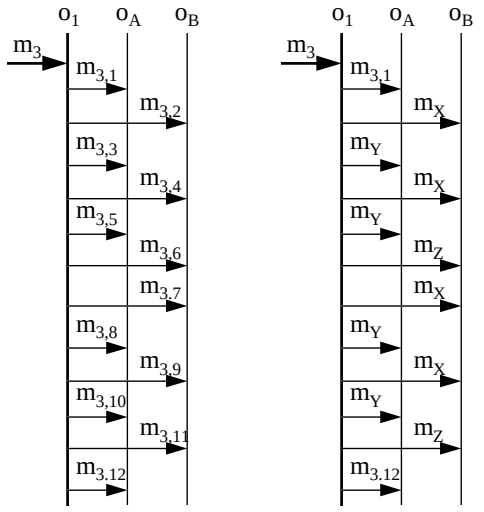

Figure 9: *Scenarios with repetition.*

We will assume that:

- $m_{3,2} = m_{3,4} = m_{3,7} = m_{3,9} = m_X$
- $m_{3,3} = m_{3,5} = m_{3,8} = m_{3,10} = m_Y$
- $m_{3,6} = m_{3,11} = m_Z$

These assumptions are made clear in scenario 3' of the figure, where we have introduced the m_x, m_y and m_z messages. Furthermore, we will assume that the preconditions on all m_X messages coincide.

Below we show two, possible conjecturings of method bodies based on scenario 3.

```
oA.m3,1;
while pre-situation(mX)
do begin
      oB.mX;
      oA.mY;
   end;
oB.mZ
while pre-situation(mX)
do begin
      oB.mX;
      oA.mY;
   end;
oB.mZ;
oA.m3,12
```

```
oA.m3,1;
while pre-situation(mX)
do begin
     while pre-situation(mX)
     do begin
          oB.mX;
          oA.mY
        end;
     oB.mZ
   end;
oA.m3,12
```

The one to the left is the result if we grap the first encountered repetition in the systematic scanning of the scenario. The one to the right will be the result if we grap the longst repetition of the scenario, and threre after, recursively, apply the same scanning in the body of the loop. In DYNAMO we search for the longest repetition first. Thus, DYNAMO will deliver the method body to the right. Notice, by the way, that we generate the same loop control condition in the outer and inner loops. This is a reminiscence of a more general problem of applying the pre-situations as loop control condition (discussed next).

The use of the precondition P of the first repeated message as the loop control condition is problematic in general. The situation before $m_{3,2}$ and $m_{3,4}$ in figure 8 may be different, but still we may want to iterate. The problem is that we cannot easily identify the *loop invariant* relevant part of the assertion. P reflects, in principle, the total, computational situation before $m_{3,2}$ and $m_{3,4}$. Consequently, the pre-situations and post-situations, as used in the messages of the DYNAMO scenarios, are not ideal as the basis for discovering loops.

As another caveat, the derivation of loops from a complete or incomplete repetition can only represent a guess. We may very well encounter situations where we conjecture a loop, but semantically we simply had to do the same thing (exactly) twice.

From an algorithmic point of view the challenge here is to determine the longest repetition among the sub-scenarios of a scenario; subsequently to identify an incomplete repetition at the end of a scenario. Some of the details can be found in appendix A.3.

4 Deriving classes.

In this section we will discuss the derivation of a class C from a set of scenarios which involves objects that are instances of C. The discussion will include the following aspects of a class:

- Class protocol, including visibility issues.
- Inheritance from other classes.
- Details about attributes.
- Details about methods.

Notice here that the derivation of method bodies already has been treated in depth in section 3.

Scenario-based dynamic models describe the interaction among objects. In DYNAMO objects are characterized by

- a unique object name,
- the name of the class, and
- a so-called object keeping.

The sole purpose of the classes is to relate the set of objects which are instances of the same class and as such share a number of properties. In the current version of the modelling language there are no further information available on the classes.

The object keeping specify how objects are related to their context. Objects may be available at *global* basis, *local* to a method behind a message, passed around via parameters and function results (such object are called *floating*), *part of* another object, or *associated from* another object. For more details on the dynamic modelling language see [6].

4.1 Deriving information about the class protocol.

The class protocol is the set of properties (methods and attributes) which are available to clients of the class.

Information about method signatures (names and formal parameters) and method comments can relatively easy be extracted from the messages in the scenarios. If there, somewhere in the set of scenarios, exists a message case of m in C there must exist a method M corresponding to m in C, or in one of the superclasses of C. We conjecture that M is private if the sender and receiver of m are identical objects[5] for all available messages m in the scenario set. This is a guess based on the evidence from the scenarios.

The attributes (instance variables) of the class C can be found be locating the following object provisions of the dynamic model:

1. An object provision which provides an object X (say of class D) as *part of* Y, and furthermore assuming that Y is an object of class C.
2. An object provision which provides an object X (say of class D) as *associated from* Y, and furthermore assuming that Y is an object of class C.

In the first case we can conclude that class C aggregates a D-part, and in the second that C associates a D-object.

We are, in addition, interested in infering the subset of the attributes being public. To that end we use the heuristic that an attribute X is private it is provided in a context of a message to a C-object. Objects provided in other contexts are estimated as being public, thus contributing to the class protocol of the class C. Again, these are guesses based on the evidence provided in the set of scenarios.

[5] In more pratical terms, any m message is sent to self, the current object.

4.2 Deriving information about class inheritance.

One of the interesting questions is whether, or to which degree, information about inheritance can be derived from a set of scenarios. Inheritance between classes is specified at class level. Our starting point is a set of objects, messages between the objects, and association and aggreation relations among the objects.

Given the informations about objects from above, it turns out to be difficult to extract knowledge about the inheritance relations between the underlying classes. It might, however, be possible to compare classes with respect to their class protocols. Let us briefly develop a concrete proposal along these lines.

Let C_1 and C_2 be two different classes, and let P_{C_1} and P_{C_2} be the class protocols of C_1 and C_2 respectively. We may conjecture that C_2 inherit from C_1 if $P_{C_1} \subseteq P_{C_2}$.[6] This is based on the heuristics that a subclass (C_2) extends the set of features which are inherited from its superclass (C_1). This is typically case, but not necessarily true in all object-oriented programming languages (Eiffel being one of the exceptions).

In the current version of DYNAMO we do not attempt to infer inheritance relationships among classed based on the ideas from above. The main reason is that we believe that inheritance should not be discoved from accidential coincidence (in the sense from above) of class protocols. Rather, inheritance should be formulated directly by the designer. Given our focus on pure dynamic models, this is not possible in DYNAMO. However, in a future version of DYNAMO we may want to specify the dynamic models and part of the static models side by side.

5 Tool support in DYNAMO.

The DYNAMO tools[7] make it possible to create and explore dynamic models.

The overall message structure of a scenario may be created and edited in the *interaction diagram editor*. In this editor, scenarios are shown as UML-like sequence diagrams (where vertical lines represent objects and horizontal lines represent messages between objects). Besides the interaction diagram editor there are a number of browsers via which it is possible to edit the details of dynamic models, messages and objects. The *dynamic model browser* allows editing of the initial scene and the message structure of the scenarios. (As such, the dynamic model browser and the interaction diagram editor are overlapping). The *message browser* allows editing of the details of a single message. The *object browser* allows editing of the details of an object.

The result of a program synthesis is shown in a DYNAMO *static model browser*. The static model browser may be activated on a dynamic model, or on a single scenario in the model. Figure 10 shows an example of the static model browser. The class list of the browser enumerates the classes of which there exists instances in the analyzed dynamic model. When selecting a class in the class list the list of the synthesized methods are shown in the method list. When activating one of the buttons "Generate class" or "Generate method" a the synthesized program element is shown in a new text window.

It is possible to filter the method list by selecting/de-selecting items in the "Methods to list" section of the browser. We may, for instance, decide that we only want to deal with public methods which contains selective or iterative control structures in the synthesized bodies.

[6] In an implementation of this rule we have to decide when two methods are equal with respect to the subset test. This may involve identical names and identical parameter lists.

[7] It is possible to see examples of all the DYNAMO tools via the World Wide Web on `http://www.cs.auc.dk/~normark/dyn-models/tool-tour/all.htm`.

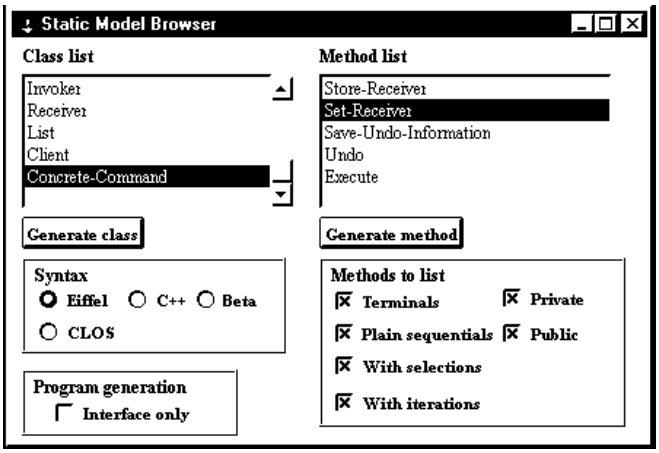

Figure 10: *The static model browser.*

It is, in addition, possible to chose the concrete syntax and the level of abstraction which is used to present the result of the synthesized classes and methods. We support the syntaxes of a wide variety of different object-oriented programming languages in order to make the designers more comfortable with the result of the synthesis process.

6 Similar work.

To appear in a later version of the paper.

7 Conclusions.

The goal of the DYNAMO project is to conclude on the hypothesis which states that *dynamic models may play an important role as the "first model" in the object-oriented design process.*

The goal of the work discussed in this paper is to conclude on the sub-hypothesis which states that *outlines of classes and methods can be synthesized from a DYNAMO dynamic model.* The confirmation of this hypothesis will clearly contribute to the main hypothesis of the project. (If we are able to derive parts of the static model from an appropriate dynamic model it is more realistic to take a dynamic model as our starting point).

The DYNAMO project is still in progress, and it is not possible to come forward with any definitive conclusions yet. Here we will therefore restrict ourselves to a discussion of our approach and compare it briefly with similar approaches.

Synthesis of program-like descriptions have been attempted in several contexts. In a paper from the mid seventies Biermann et al. describe how to construct programs from example executions [1]. Based on a number of condition/instruction traces the algorithm of Biermann et al. generates a minimal program, which is able to execute all the traces. The algorithm basically searches through all possible programs which can be formed on the given traces. The order of the search guarantees that the smallest possible program will come out as the result. The program is represented as a transition graph, where the nodes are instructions and the edges are conditions. Kosmimies et al. discuss how to utilize the Biermann approach to synthesize state machines from

scenarios [3]. Notice that a state machine can be regarded as transition graph too. It may be a challenge in its own right to generate conventional object-oriented program text from such transition graphs.

In contrast, we generate classes and methods in a programming language like notation from a set of DYNAMO scenarios. Thus, we emphasize that the result of the synthesis should comply with the usual object-oriented program structure in terms of classes, class relations (inheritance, associations, aggregation), and methods. Following our approach, we carry out a number of abstract syntactic transformations on a collected subset of the scenarios in order to synthesize the method bodies. We are looking for a number of patterns in the set of relevant of scenarios. We are confident that the existing patterns are useful, but several additional patterns may exist. This approach is both a strength (because of simplicity) and a weakness (because of its inherent incompleteness).

It may be possible to apply the synthesis techniques of the Biermann and the Koskimies groups in our setting as well. This may be a theme in our continued research.

Besides the synthesis of method bodies we also care about class protocols (visibility of attributes and methods) and relations among classes. We have demonstrated that it is possible to estimate the visibility of methods in a class from the scenarios. It is harder to infer contributions to the generalization/specialization structures among classes from the scenarios. In addition, we have in this paper questioned whether it is worthwhile to to so at all.

A Algorithms and datastructures

In this appendix we will describe the algorithmic aspects of our work. DYNAMO is implemented in Common Lisp and CLOS using Allegro Common Lisp for Windows (on a PC platform). For the sake of this appendix, however, we will describe the algorithms and data structures in a language neutral way.

A.1 The abstract grammar

As a data structure, a dynamic model is probably best understood as a tree derived from the abstract grammar shown below:[8]

```
<dynamic-model> ::= <initial-scene> <scenario>-list
<scenario> ::=    <object> <activation>
                       <object-provision>-list
                       <scenario>-list
                  <result>

<activation> ::=  <operation> <actual-parameter>-list
                  <pre-situation>
                  <general-understanding> <specific-understanding>

<object-provision>  ::= <class> <object-id> <keeping> <object-understanding>
<keeping>  ::=  <keep-as-part> | <keep-as-associated> | <global> |
                <local> |<floating>
<result>     ::=  <effect> | <object-id> | <object-provision>
<initial-scene>      ::=  <object-provision>-list
<actual-parameter> ::= <object-id> | <object-provision> | <informal-parameter>
```

[8] The <activation> constructor might as well be folded into the scenario constructor.

The meaning of syntactic categories is discussed in section 2 of this paper. The Syntactic categories (nonterminals) without a left-hand-side are either strings/symbols or just atomic terminal informations. The syntactic categories with a left-hand-side definition corresponds to either CLOS classes that aggregates other syntactic categories (constructors) or CLOS (super)classes with a number of specializations (alternatives).

In addition to the abstract grammar of the dynamic modelling language we introduce a simple language for *conditional regular expression*. This term covers regular expressions in which each alternative in a selection is controlled by a condition, and in which iteration is controlled by a single condition (repeat as long the the condition holds). The terminal elements of conditional regular expressions are objects of the syntactic categori `scenario`.

```
<regular-expression> := <regular-sequence> | <regular-selection> | <regular-iteration>
<regular-sequence> := <regular-element>-list
<regular-selection> := <regular-element>-list  <predicate>-list
<regular-iteration> := <predicate> <regular-element>
<regular-element> := <scenario> | <regular-expression>
```

In a `regular-selection` there must be a `regular-element` for each predicate, and vise versa. Thus, the `regular-element` list and the `predicate` list must have the same lengths.

It will be made clear below how we use conditional regular expressions as a representation of a method body that consists of several messages cases of a message m in a class C.

A.2 Conditional method derivation

In this section we will describe the algorithm behind the derivation of method bodies with conditionals, as discussed in section 3.2. The starting point is a dynamic model in terms of an initial scene and a set of scenarios. The input is a message m and a class C. The desired result is a method body of M. The function with this input-output characteristics is called `synthesize-method-body`:

```
synthesize-method-body(m: method-name, C: class-name): regular-expression
let mc-list = collect all message cases for m in C in the scenario set
    mc-list-1 = the filtering terminal messages out of mc-list
    mc-list-2 = if empty(mc-list-1) then list(first(mc-list)) else mc-list-1
    mc-list-3 = the filtering duplicates out of mc-list-2
    mc-list-4 = map propagate-post-to-pre on mc-list-3
    reg-exp-1 = regularize(mc-list-4)
in
  simplify-regular-expression(reg-exp-1)
end.
```

Recall that a message case (a scenario) is terminal if it contains no submessages (its scenario-list is empty), see 3.1. If all messages cases for m in C are empty we see that `mc-list-2` becomes the list of one of the terminal message cases.

The function `propagate-post-to-pre` overwrites a trivially true pre-situation with the predecessing post-situations, if the post-situation is non-trivival. In terms of the notation from section 3.2, pre-situation($m_{i,j}$) becomes post-situation($m_{i,j-1}$) in case the presituation of $m_{i,j}$ is trivially true. This is convenient because it allows the designer to state the assertions where it is most natural in a scenario, without having too much redundancy between a pre-situations and the post-situations of the previous message in the scenario.

The function `regularize` transforms a list of scenarios to a regular-expression. In case the list contains more than one element a regular-selection is returned, which selects among the sequences of sub-scenarios of the scenarios parameter.

```
regularize(scenarios: list of message-cases): regular-expression
if length(mcl) = 1
then make-regular-sequence(scenario-list(first(mcl)))
else let sequences = map (make-regular-sequence on scenario-list of each scenario) on scenarios
         predicates = map (pre-situation of activation of each scenario) on scenarios
     in make-regular-selection(sequences, predicates)
```

The function `simplify-regular-expression` is central because it is the function that extract a common prefix or a common semi-prefix from a set of scenarios, represented as a regular-selection. Thus, `simplyfy-regular-expression` is the function which transforms the body of the method to a form that is more implementation-like:

```
simplify-regular-expression(sel: regular-selection): regular-expression
let sel-1 = remove-empty-alternatives(sel)
    prefix = common-prefix(sel-1)
in if length(prefix) > 0
   then make-regular-sequence(prefix, remove-prefix(prefix,sel-1))
   else simplify-regular-expression-1(sel1)
```

We first remove empty alternatives from the `regular-selection`. The function `common-prefix` identifies the longest prefix of the scenarios in `sel-1`. Let us assume that `sel-1` is selection among n sequences s_1, s_2, ..., s_n. Each of sequence s_i consists of the "elements" $s_{i,1}$, $s_{i,2}$, ... s_{i,j_i}. It is straightforward to make a recursive function `common-prefix-2` which finds a common prefix of two sequences (not shown here). First we find the common prefix, p_1, of s_1 and s_2. Next we find the common prefix of p_1 and s_3, and so on until all n sequences have been processed.

If a common prefix is found, we form a sequence of the prefix and the remaining regular-selection, which is returned by `remove-prefix(prefix,sel-1)`. An example illustrating this transformation can be seen on page 7, where method body [1] is transformed in method body [2].

If no common prefix is found, we call `simplify-regular-expression-1` in order to possibly find a semi-prefix.

```
simplify-regular-expression-1(sel: regular-selection): regular-expression
if number of alternatives in sel > 2
then let odd-alternative = find-odd-alternative(sel)
         other-alteratives = all-alternatives-but(odd-alternative, sel)
     in if odd-alternative
        then make-regular-selection(
               list(simplify-regular-expression(
                         make-regular-selection(other-alternatives, appropriate conditions))
                    odd-alternative)
                appropriate conditions)
        else sel
else sel
```

We first identify the possible single alternative which hinders the identification of a common prefix. If such a one exists we form a new selection between the "odd alternative" and the remaning case. In the remaining case we activate `simplify-regular-expression` recursively in order to extract the common prefix of the "non odd alternatives". Notice that in the function above we does not describe in precise terms how to handle the conditions of the involved

regular-selections. As an illustrative examle the function `simplify-regular-expression-1` is responsible for transforming method body [2] to method body [3] on page 8.

What is left is now to determine a possible "odd alternative" among the n alternatives in a regular selection. Let us again look at the n alternatives s_1, s_2, ..., s_n from above. We now take a closeer look at $s_{1,1}$, $s_{2,1}$, ..., $s_{n,1}$. We want the find i such that $s_{i,1}$ only appears once in the set $\{s_{1,1}, s_{2,1}, ..., s_{n,1}\}$. The scenario s_i is the "odd alternative".

Some statement about the complexity.

A.3 Iterational method derivation

In this section we will describe how we identify a repeting subsequence among the sub-scenarios of a given scenario. Let us study the scenario s, as represented by the message case mc, which sends the messages m_1, m_2, m_k. The problem is to find a repetition in this sequence of messages. The function `find-and-transform-repetitions` does the job:

```
find-and-transform-repetitions(seq: regular-sequence): regular-expression
let rep = repetition(the elements of seq)
in if rep
   then aggregate-iteration(split-sequence(seq,rep))
   else seq
```

The function `repetition` searches for a repetitional subsequence of `seq`.

```
repetition(scenarios: list of scenario): list of scenario
if empty(scenarios)
then empty-list
else let immediate-rep = prefix-repetition(scenarios)
     in if immediate-rep
        then immediate-rep
        else repetition(tail(scenarios))
```

The function `prefix-repetition` searches for a repetition starting with the first element of the list. As an example the list `(a b)` is repeated immediately in `(a b a b c a b a b c d e)`. However, the function does not return `(a b)`, but instead `(a b a b c)` because this list is a longer prefix repetition.

```
prefix-repetition(scenarios: list of scenario):  list of scenario
find-prefix-repetition(list(first(scenario)), tail(scenario)).

find-prefix-repetition(prefix, rest: list of scenario): list of scenario
if empty(rest)
then empty-list
else if prefix-repeated-immediately(prefix,rest) and
        not find-prefix-repetition(append(prefix, first(rest)), tail(rest))
     then prefix
     else find-prefix-repetition(append(prefix, first(rest)), tail(rest)).
```

The function `prefix-repeated-immediately` is a straightforward function (not shown here) which tests whether its first parameter coincide with the first "few" elements of the second parameters. In the actual DYNAMO implementation, `find-prefix-repetition` handles the additional case of *incomplete repetitions* (as discussed in section 3.3).

Now having explained how to find the wanted repetition, we must deal with how to split the original sequence into a prefix, a repeating part, and a suffix. This is the job of the function

`aggregate-iteration`. But first `split-sequence` is invoked on the original sequence and located prefix.

```
split-sequence(seq,rep: list of scenarios): three lists of scenarios
let prefix = the elements before rep occurs in seq
    rep = the elements of the repetitions, as many times as possible
    suffix = the remaining elements of seq
in (prefix, rep, suffix).

aggregate-iteration(prefix,repetition,suffix: list of scenario): regular-sequence
make-regular-sequence(
   append(find-and-transform-repetitions (make-regular-sequence(prefix))),
          make-regular-iteration(
              find-and-transform-repetitions(make-regular-sequence(repetition)),
              appropriate condition),
          find-and-transform-repetitions (make-regular-sequence(suffix))).
```

Again, we do not explain in details how find the "appropriate conditions". As an interesting detail, notice how the search for repetitions is activated recursively in the function above.

As an example of an application of find-and-transform-repetions please take a look at method body [4] on page 11.

Some statement about the complexity.

Finally, we have to deal with how `find-and-transform-repetitions` must be called from `simplify-regular-expression`, which we discussed in section A.2. We will not go into these details in the present paper. However, it should be noticed as a general remark that the combination of the supported transformations very well may turn out to be an issue of great importance if we look for other patterns than common prefixes, common semi-prefixes, and repetitions.

References

[1] A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, SE-2:141–153, 1976.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.

[3] Kai Koskimies and Erkki Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software - Practice and Experience*, 24(7):643–658, July 1994.

[4] Kai Koskimies, Tatu Männistä, Tarja Systä, and Jyrki Tuomi. SCED: A tool for dynamic modelling of object systems. Technical Report A-1996-4, Department of Computer Scienece, University of Tampere, Finland, 1996.

[5] Bertrand Meyer. *Eiffel the Language.* Prentice Hall, 1992.

[6] Kurt Nørmark. Towards an abstract language for dynamic modelling in object-oriented design. In *TOOLS'97*, 1997.