Synthesis of Program Outlines from Dynamic Models in DYNAMO

Kurt Nørmark
Aalborg University
Denmark*

April 23, 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.

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 properties of objects and messages are concentrated to shared properties

*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.

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 development efforts.

It is one of our sub-hypotheses that significant parts of a static model can be synthesized from the information which is present in a dynamic model, as supported by DYNAMO. In this paper we will discuss to which degree it is possible to generate outlines of 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. These abstraction barriers are carefully protected in the static, class-based models as well as in the source programs of implementation phase. 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. In that way a scenario gives a useful *holistic view* of the ways objects interact with each other.

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 can be used as skeletons in a refinement process on the way towards the final source program.

In this paper we will first describe the scenario concept. This is the foundation of the rest of the paper, because the synthesized classes and methods depend exclusively on the information which is present in the scenarios. Following that we discuss how to synthesize various aspects of classes and methods. Most of discussion is oriented towards synthesis of method bodies with either selection or iteration.

2 The scenario concept in DYNAMO.

The scenario concept of DYNAMO is simple and straightforward. The main reason is that we keep the scenarios clean from control mechanisms such as selection and iterations. This is in contrast to the algorithmic scenarios, as found in [3].

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 receiving object (called *receiver*(m)), some actual parameters, a pre-situation assertion, an informal understanding of the message, a list of object provisions, a list of sub-messages to other objects, a result (describing the effect of the message or the object which is returned by the message), and a post-situation assertion.

The pre-situation and post-situation are assertions that describe the situations at some given points in time during a scenario. The assertions are informal elements which describe the situation per se, at the point where they is located in a scenario.

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. An object provision claims 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. In addition, the class name of an object is also registered. In the current version of DYNAMO all objects are passive.[1]

The *object keeping* specify how objects are related to their context. Objects may be available on *global* basis, *local* to a message, passed around via parameters and function results (such object are called *floating*), *part of* another object, or *associated from* another object.

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

3 Class Synthesis.

In this section we will discuss the synthesis of a class C from a set of scenarios which involves objects that are instances of C. The discussion will include visibility issues (private vs. public attributes/methods), inheritance issues, attributes, and methods.

Each example of a message m to some C-object (some object of class C) will hereafter be called a *message case for m in C*.

As a matter of notation, lower case letters will in the sequel be used for elements of the dynamic model (objects and messages) whereas upper case letters will be used for elements of the static model (classes and methods).

Synthesis of the class protocol

The class protocol is the set of properties (attributes and methods) which are available and relevant 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 for 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[2] for all available messages m in the scenario set. This is a guess based on the evidence from the scenarios. The thoughts behind the guess is that "if all messages m goes to the current object, there is no reason to include the underlying method M in the class protocol of C".

Later in this section we will discuss how to synthesize outlines of method bodies.

[1] It will be an obvious extension also to consider active objects. However, this will affect the entire dynamic modelling language. We consider such an extension as a natural next step in the research process.

[2] In more practical terms, any m message is sent to *self*, the current object.

Synthesizing class inheritance information

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 $P_{C_1} \subseteq P_{C_2}$.[3] 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 [4] 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.

Synthesis of simple method bodies

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 to base M, only contains a single message case *mc* for m in C. I.e., the entire set of scenarios only contains one example of an m-message to a C-object. Consequently, the method needs to be generated from a single example.

In the most elementary situation, the message case does not have any sub-messages. We say that the message is *terminal*. This is either because no additional communication is necessary in order to fulfil 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 objects) and/or to itself. It is possible, in a straightforward way, to transform the available information about *mc* into an outline of the method M. The method name comes from the name of the message, the formal parameter list of the method is constructed from the actual parameters of the message (where we know the classe of the involved objects), the method comment stems from the message understanding, and the list of commands in the method body comes from the list of direct sub-messages of m.

Synthesis of method bodies with selection

A more interesting situation arises when two or more message cases contribute to the method we are synthesizing from scenarios. The challenge is here how to subsume a number of examples

[3] 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.

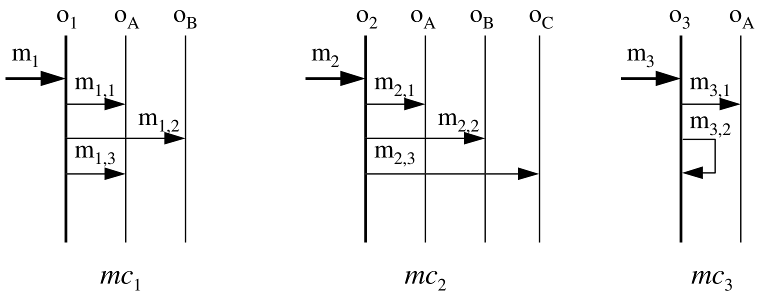

Figure 1: *Three message cases for m in a class C.*

(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

$$\mathrm{m}_{i,1},\ \mathrm{m}_{i,2},\ \mathrm{m}_{i,n_i}$$

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

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) to other objects, have pre-situations and post-situations. Thus, the message case mc_i gives rise to the following "Hoare statements" for $i = 1, ...n$:[5]

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

Here P_i = *pre-situation(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 are subclasses of C and that C contains a virtual method which "covers" all the message cases. 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 1, the body of the synthesized method M becomes the following construct:

[4]In the figure, the following equations relate the notation in the figure to the general notation: $\mathrm{m}_1 = \mathrm{m}_2 = \mathrm{m}_3 = \mathrm{m}$, $n_1 = n_2 = 3$, $n_3 = 2$, $o_{1,1} = o_{2,1} = o_{3,1} = o_A$, $O_{1,2} = o_{2,2} = o_B$, and $o_{2,3} = o_C$.

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

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.$m_{3,2}$

However, this 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.

In order to illustrate this we can assume that $m_{1,1} = m_{2,1} = m_{3,1}$ in figure 1. In this context, two messages are equal if they have the same name and if their receiver objects are identical.[6] 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 synthesis of the method looks like:

$o_A.m_{*,1}$;
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 1, 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:[7]

$o_A.m_{*,1}$;
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

[6] 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.

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

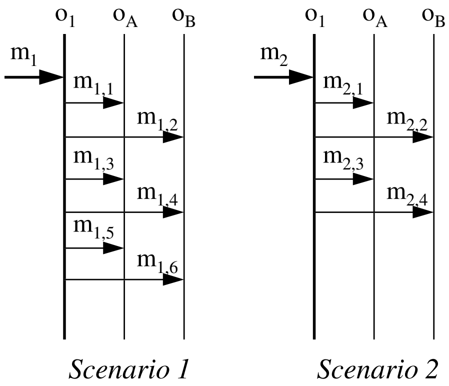

Figure 2: *Two scenarios with repetitions.*

pre-situations of $\mathrm{m}_{*,2}$. In practical situations this does not lead to any difficulties, because the designer is not likely to specify both a pre-situation of $\mathrm{m}_{i,2}$ and a post-situation of $\mathrm{m}_{i,1}$.

Synthesis of method bodies with iteration

The main point in the discussion until now has been the synthesis of a single method with selection from multiple message cases. We will now direct our interests towards a single scenario with patterns of repetition. Let us assume that

- $\mathrm{m}_{1,2} = \mathrm{m}_{1,4}$
- $\mathrm{m}_{1,3} = \mathrm{m}_{1,5}$
- *pre-situation*($\mathrm{m}_{1,2}$) = *pre-situation*($\mathrm{m}_{1,4}$)

in scenario 1 of figure 2.

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

$\mathrm{o}_\mathrm{A}.\mathrm{m}_{1,1}$;
while *pre-situation*($m_{1,2}$)
do begin
    $\mathrm{o}_\mathrm{B}.\mathrm{m}_{1,2}$;
    $\mathrm{o}_\mathrm{A}.\mathrm{m}_{1,3}$
  end;
$\mathrm{o}_\mathrm{B}.\mathrm{m}_{1,6}$

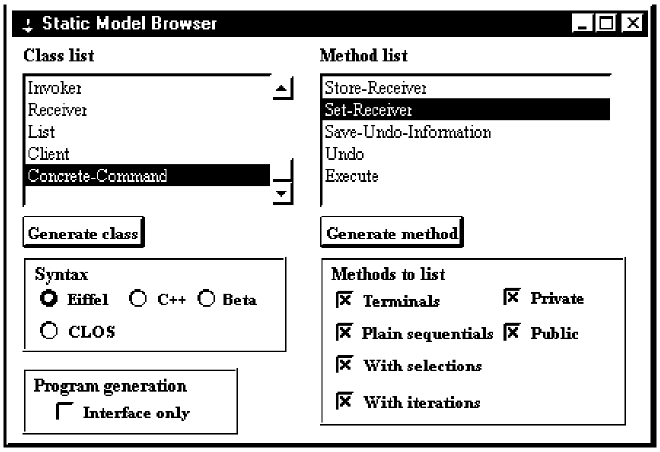

Figure 3: *The static model browser.*

The conjecture of the loop above is based on the observation of a *complete repetition.* In DYNAMO we also support conjectures of loop based on *incomplete repetitions*, but only at the *end* of a scenario. Scenario 2 in figure 2 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 same. This leads us to conjecture that $m_{2,4}$ is the start of a repetition 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 2.

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

4 Tool support in DYNAMO.

The DYNAMO tools[8] 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

[8]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`.

model browser may be activated on a dynamic model, or on a single scenario in the model. Figure 3 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.

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.

5 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 [2]. 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). In this paper we have described the outcome of these transformations. In a more complete companion paper [5], more examples are given (including examples taken directly from DYNAMO). Moreover, we describe the algorithmic principles behind the transformations in the companion paper.

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.

References

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

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

[3] 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.

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

[5] Kurt Nørmark. Deriving classes from scenarios in object-oriented design. Forthcoming paper, 1997. Preliminary version available on http://www.cs.auc.dk/~normark/dynamo.html.

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