Towards an Abstract Language for Dynamic Modelling in Object-oriented Design

Kurt Nørmark
Department of Computer Science
Aalborg University
Fredrik Bajers Vej 7
DK-9220 Aalborg
Denmark
Email: normark@cs.auc.dk

Abstract

Modelling with objects at a concrete level is attractive as a supplement to modelling with classes. The former is known as dynamic modelling and the latter as static modelling. This paper is about a language for describing dynamic models. The language is defined at an abstract level, thus allowing us to concentrate on the underlying conceptual issues rather than more superficial ones such as diagrammatic notations. Our dynamic modelling is supported by a set of tools called DYNAMO. In this paper we concentrate mainly on the concepts of the abstract modelling language, but we do also touch on the DYNAMO tools.

1 Introduction

In the area of object-oriented design we generally distinguish between two kinds of models, static and dynamic models. Static models are oriented towards classes and class relationships. In general, static models emphasize the structural properties of a design. Dynamic models deal with objects and object relationships. Dynamic models emphasize the process and temporal properties of a design.

In this paper we will first discuss a number of issues related to dynamic models, which motivates our approach. In section 3 we describe the concepts of a particular dynamic modelling language. To make our discussion more concrete we also give an example of a dynamic model. In section 4 we briefly discuss the DYNAMO tools for construction and exploration of dynamic models. In section 5 we discuss possible extensions of the modelling language together with future work. Finally, in section 6 we give an overview of similar work.

2 Starting point and motivation

Several kinds of dynamic model diagrams are known from the OOD literature [Booch, 1994, Rumbaugh *et al.*, 1991, Jacobson *et al.*, 1992]. Using UML (version 1.0) terminology [Rational Software Coorporation, 1997], *sequence diagrams* show the interaction among a set of objects in temporal order. *Collaboration diagrams* show the interaction among a set of

[0]

objects as a graph. Both sequence diagrams and collaboration diagrams deal with *scenarios*, i.e., "a particular series of interaction among objects in a single execution of a system" [Rational Software Coorporation, 1995]. In the rest of this paper we will call UML-like sequence diagrams for *interaction diagrams*.

In the work described in this paper we stick to a scenario-based design approach. As a limitation of the current work, we deal exclusively with uni-sequential scenarios.

Scenario-based models are different from *state machine* models. State machines are dynamic models in the sense that they deal with the evolution of object state as a function of events in the object environment. State machines are clearly rooted at a deeper semantical level than scenarios-based models. However, state machine models really focus on the behavior of an arbitrary instance of a class. It takes a number of state machines to specify the mutual interaction among a set of objects. Thus, state machine models give a fragmented understanding of the overall behavior of an object system. This is in contrast to scenarios, which convey a more holistic description of the interaction among the objects.

Although an obvious duality exist between static and dynamic model concepts, we often encounter design situations where static models are inadequate for expressing some central aspects of a design. This is typically the case if the patterns of object interactions play a more important role for the design than the static relations among classes, attributes and methods. In such situations it is common to supplement the static model with one or more dynamic models. However, the approach recommended in the present work is more radical. Our starting point is the following hypothesis:

> *Programmers think in terms of objects, object relations, and object interactions during the creative phases of the design process.*

As a consequence of this we find that it is natural to formulate dynamic models *before* static models in order to capture the intended design at a concrete and tangible level.

Despite the emphasis on dynamic models early in the design process, static models are still important. A static model can be seen as an abstraction of a dynamic model. In DYNAMO we make considerable efforts to infer aspects of a static model from a dynamic model. We touch on that later in the paper (see section 4), but for a more complete discussion please consult [Nørmark, 1997a].

Here it is also relevant to point out the similarity between our approach and use case driven object-oriented design. The purpose of use cases is to capture the requirements to a system. According to Jacobson et al. a use case is a behaviorally related sequence of transactions in a dialogue between an "actor" and a "system" [Jacobson *et al.*, 1992]. A scenario can be thought of as a implementing part of a use case [Graham, 1996].

During the object-oriented design phase the diagrammatic means of expression dominates. One reason is undoubtedly that diagrams provide a general overview at some relatively high level of abstraction. This is in contrast to the programming phase where the textual linguistic means of expression dominate, not least because of the many details that must be dealt with at this level of the development process. In general, the means of expression play a crucial role for the human perception of a model. However, in the present work we de-emphasize the means of expression in the starting point. Instead we develop an *abstract language* based on the concepts which we find useful for expressing dynamic models. We call it an abstract language because we are not, at the outset, concerned with definition of the concrete syntax of the language, nor any other particular means of expression. A dynamic design model is represented as a data structure which is akin to abstract syntax trees as familiar from the representation of programs relative to an abstract grammar.

We see a number of advantages of basing this work on an abstract dynamic modelling language:

1. We may absorb a number of details in the data structure that represent a model without necessarily cluttering the external appearance of the model. There is no notion of a canonical form of a design, during which all possible details must be formulated. The designer enters a dynamic model via some editing tools. Some of these tools filter the design descriptions so that only certain aspects of the design need to be dealt with at a given time.

2. As a special case of the first item, we can capture intuitive descriptions of objects and interactions, and make these descriptions available to the designer via the editing and browsing tools.

3. We can bring life to the dynamic model, via interpretation or animation of the internal design representation (the data structure mentioned above). With this we may reflect the true nature of a dynamic model, as something 'alive'.

4. The abstract representation can be shown, via a presentation tool, in a variety of different ways to the designer: as a diagram or graph, or as text in a formal language, or by other means.

Most of observations above are direct consequences of the structure editing approach [Nørmark, 1996b, Nørmark, 1987] used in DYNAMO.

3 The abstract language

In this section we describe a scenario-based dynamic modelling language. The purpose of the scenarios is to serve as the *first and primary model* in the object-oriented design process, in particular in design situations that are "difficult" to deal with exclusively through static models (class diagrams).

This paper presents work in progress. As a consequence, the abstract design language has not yet found its final form. In section 5 we will discuss the planned extensions and modifications in the next version of the language.

3.1 Central concepts

The main concepts of our abstract language are objects, scenes, object-provisions, scenarios, and messages.

An *object* is an encapsulation of state with identity. This definition is similar to the object concept from object-oriented programming languages. We do not represent the data of an object in terms of fields—this would be an overkill in a scenario-based design language. Keeping and maintaining a detailed object state would bias towards actual calculations, but this is not an issue during the design phase. In the current version of the language, only indirect and informal means are used to describe the state of objects. However, we do represent how an object is related to other objects. The current version of the language can capture how an object is part of or associated with another object. The class of an object is also registered. The class itself does not exist as a (meta)object in the dynamic model. The sole purpose of classes is to relate the set of objects which are instances of the same class, and as such share a number of properties.

A *scene* is the set objects relevant to a scenario at some given point in time. Objects enter the scene via a mechanism called *object provision*. Each object in a scenario must be provided exactly once. Object provision is a convenient mechanism that states the relevance of an object at exactly the time it is needed by the designer. In some cases a provided object may actually have existed for 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 provision time. In these situations, object provision is identical to object creation. Either way, an object provision claims the existence of an object that possesses certain relationships to the objects already existing on the scene. As a simple and practical convention, each object on the scene has a unique name. Other kinds of object handles (via references or dot notation) are considered irrelevant for design purposes.

The life-time of an object and its relationships to other objects are represented as *object keepings*. The possible object keepings in the current version of the language are:

- *Global*: The object can be used in the rest of the scenario relative to the place of its provision.
- *Local*: The object can be used in the operation *op* in which it is provided. The object may be passed as parameter to operations called from *op*. However, a local object ceases to exist when *op* returns.
- *Floating*: The object is local (in the sense from above) to the operation *op* in which it provided. In addition, the object may be returned from *op* as the result. In that case the object survives as long as we pass it around (via parameters and as function results).
- *Part-of*: The object is part of another object X. X may be global, local, floating, or part of or associated from a third object.
- *Associated-from*: The object O is associated from another object X. In other words, there exists and asymmetric relation between the O and X in the direction from X to O. Thus, in a programming language the object O would be accessed as `X.fieldNameOfO`. X may be global, local, floating, part of, or associated from a third object.

A *scenario* is described in terms of a message from the “surround” to a receiver object R on the scene. (The message concept is described in detail below). In addition a scenario may, recursively, include messages from R to other objects. Thus, a scenario can be used to describe object interaction across the usual abstraction barriers of an object-oriented design. These abstraction barriers are carefully protected in the late design phase and in the implementation phase of the program development, thus adding to the decentralized and fragmented nature of an object-oriented design. However, in the early design phase it is often useful to describe the interaction across these barriers. Scenarios crossing the abstraction barriers tend to give holistic views on crucial interaction patterns among objects, and as such the scenarios may be important vehicles for understanding some of the overall elements in the chosen solution to the problem.

A *dynamic model* in DYNAMO consists of an *initial scene* and a set of scenarios. A subset of the scenarios may involve identical messages to a particular object. For instance, if we model a programming environment, there may be several scenarios which at the top level involve a message *compile* sent to a source program object. This makes sense because there may be several different outcomes of compiling a source program: compilation complete, syntactic errors, type errors, etc. Each of these is described by a particular scenario.

We use the conventional message passing metaphor for object-interaction. A *message* is passed from a sender object to a receiver object. At the top-level of the scenario there is no real sender object, although the “surround” or the designer may be seen as the sender. In addition to the receiver, it is also possible to pass parameters. Parameters may be existing objects (referred to by their names), object provisions (objects whose existence we claim

```
Send message REQUEST-COMMAND-EXECUTION to A-CLIENT
  Send message REQUEST-COMMAND-EXECUTION to AN-INVOKER
    Provide A-COMMAND as the object which has been associated from AN_INVOKER
    Send message EXECUTE to A-COMMAND
      Provide A-RECEIVER as the object on which the command ultimately is carried out
      Send message SAVE-UNDO-INFORMATION to A-COMMAND
      Result of SAVE-UNDO-INFORMATION: undo information has been saved.
      Send message DO-ACTIONS to A-RECEIVER
      Result of DO-ACTIONS: command has been effected.
    Result of EXECUTE: the effect of command has taken place.
    Send message ADD(A-COMMAND) to COMMAND-HISTORY
    Result of ADD: the parameter has been added to the list.
  Result of REQUEST-COMMAND-EXECUTION: The command has been executed.
Result of REQUEST-COMMAND-EXECUTION: The command has been executed.
```

Figure 1: *A trace of* `REQUEST-COMMAND-EXECUTION`.

at parameter-passing time), or informally-given values (of basic types that are not necessarily related to any class).

Passing a message M to a receiver R may cause the following actions to take place in the model:

1. Provision of a number of objects.

2. The passing of a number of messages from R to other objects.

3. The establishing of a *result* of M (in terms of an existing object, an object provision, an informally given value, or an informally described effect on the state of R).

The passing of messages from R to other objects in item 2 illustrates the recursive nature of a scenario with respect to message passing.

In addition, the *start situation* just after the parameter passing may be described by an assertion. Such assertions allow us to distinguish scenarios which involve identical top-level messages, as discussed above. The start situation is somewhat similar to a pre-condition, as known from programming languages, such as Eiffel [Meyer, 1988]. However, the start assertion does not express a prerequisite that must be met in order for the scenario to proceed. Rather, the assertion describes the situation which we assume is present at the given point in a scenario.

Similarly, we support an *end situation*, which is an assertion that characterize the result of a message. The end situation assertion is quite similar to an Eiffel-like post condition.

3.2 An example

We will now discuss an example of a dynamic model: the command design pattern [Gamma *et al.*, 1996]. The idea behind the command design pattern is to turn interactive commands (such as the ‘cut’ and ‘paste’ commands in an editor) into objects, and to install such command objects in invoker objects (such as in pop menu items). When a command is installed it receives information about a receiver object (such as the document being edited) that will be affected by executing a command.

The designer works with the dynamic model via a set of tools. The design itself is represented internally as a data structure. Here we will discuss the example scenarios by means of *traces* and *interaction diagrams*. A trace contains a selection of the most relevant information from a scenario.

Figure 1 and 2 shows a scenario for executing an atomic command following the ideas behind the command design pattern. The dynamic model, from which the example is taken, contains several other scenarios for command execution, such as executing a composite command.

At root level of the scenario we send the message `REQUEST-COMMAND-EXECUTION` from the surround to `A-CLIENT` object. In turn, the client object sends a similar message to `AN-INVOKER`. Via the setup, which we here assume already has taken place, the invoker is associated with an object `A-COMMAND`, which is executed by passing it the message `EXECUTE`. At the next level the two important interactions of execute are shown. First undo information is saved by the message `SAVE-UNDO-INFORMATION`, sent to the command object itself. Following that the ultimate action is carried out by sending the message `DO-ACTION` to the so-called receiver, here an object called `A-RECEIVER`. Again, the association from the command to the receiver has been established during a setup message, which we do not show as part of the current scenario. Finally, we see that `REQUEST-COMMAND-EXECUTION` adds `A-COMMAND` to an object called `COMMAND-HISTORY`. This is an object, which holds the entire history of executed commands.

In the trace in figure 1 we see how `A-COMMAND` and `A-RECEIVER` are provided during the scenario. In both cases, the objects become relevant for our purposes, and therefore we bring the objects on the scene via use of the object provision mechanism. The other objects used in the scenario are part of the initial scene of the dynamic model, and as such they are available in all scenarios.

The trace shown above does not reflect all the details of the model, as captured in the internal representation. Traces do not show the assertions (start and end situations), the class of the objects, the keepings of objects, and the message understandings (which comes in both general and specific flavors).

When working with a scenario it is very useful to know the objects on the scene at any given time during a scenario. This is valuable because these are the objects among which interaction can take place. We may ask for an exact listing of the objects on the scene at any point in a scenario. Even more useful and versatile, the scene is illustrated in the interaction diagrams via the solid lines imposed on the vertical lines representing objects. In figure 2 we see clearly that `A-COMMAND` and `A-RECEIVER` enter the scene during the scenario.

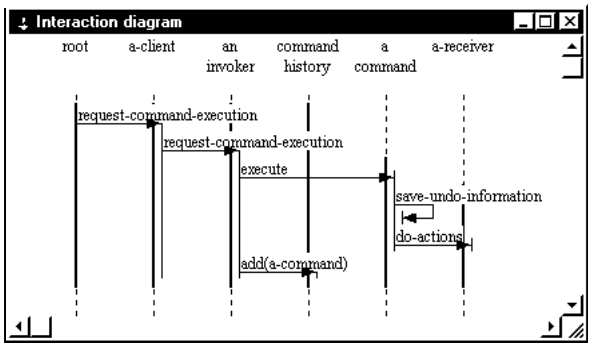


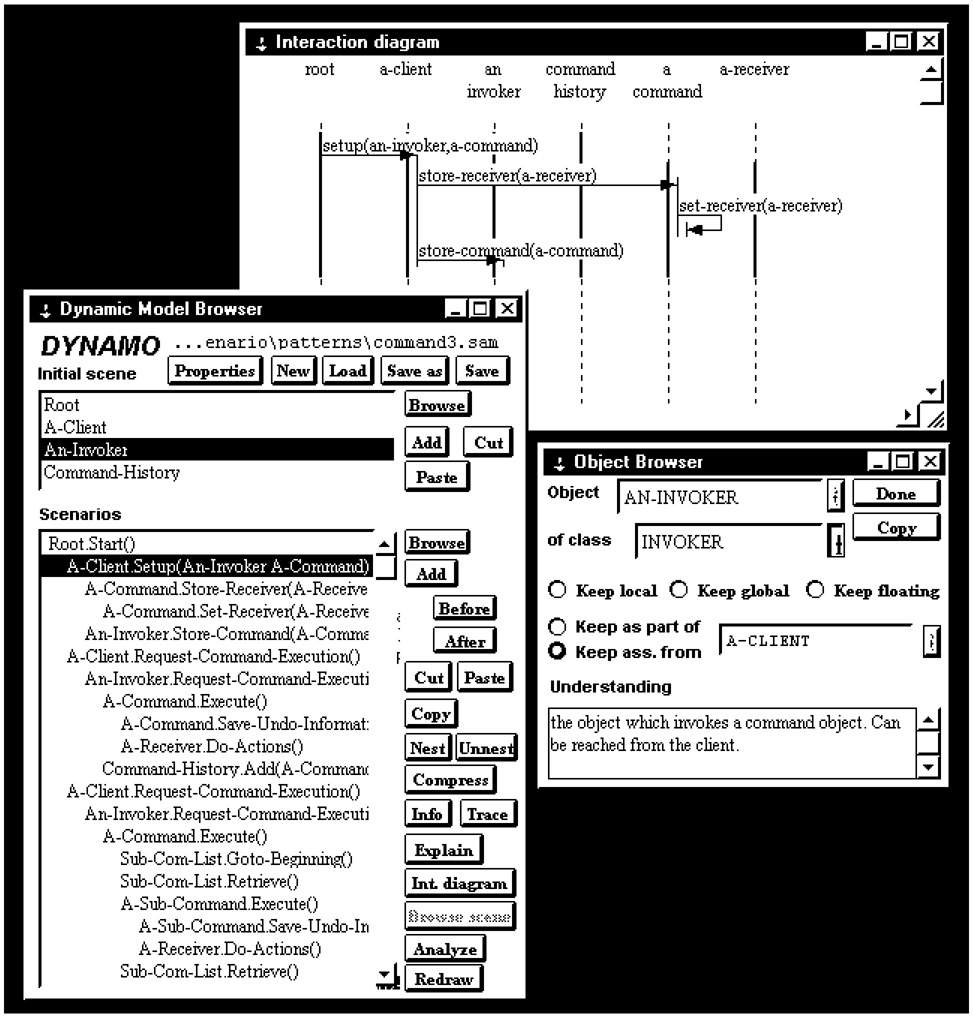
Figure 2: *An interaction diagram similar to the trace in figure 1.*

4 Tool support

The DYNAMO tools[1] realize what we think of as a *dynamic medium* in which we can create and explore a dynamic object-oriented design model [Nørmark, 1997b].

There are three tools in DYNAMO that act as browsers through which we can create a dynamic model. The *dynamic model browser* allows editing of the initial scene and the message structure of the scenarios. As such the dynamic model browser serves as the tool that keeps track of the overall elements of a dynamic

[1]It is possible to see examples of the DYNAMO tools via WWW on `http://www.cs.auc.dk/~normark/-dyn-models/tool-tour/all.html`.



Root
A-Client
An-Invoker
Command-History

Root.Start()
A-Client.Setup(An-Invoker A-Command)
A-Command.Store-Receiver(A-Receive
A-Command.Set-Receiver(A-Receive
An-Invoker.Store-Command(A-Comma
A-Client.Request-Command-Execution()
An-Invoker.Request-Command-Executi
A-Command.Execute()
A-Command.Save-Undo-Informat:
A-Receiver.Do-Actions()
Command-History.Add(A-Comman
A-Client.Request-Command-Execution()
An-Invoker.Request-Command-Executi
A-Command.Execute()
Sub-Com-List.Goto-Beginning()
Sub-Com-List.Retrieve()
A-Sub-Command.Execute()
A-Sub-Command.Save-Undo-In
A-Receiver.Do-Actions()
Sub-Com-List.Retrieve()

Object: AN-INVOKER
of class: INVOKER
○ Keep local ○ Keep global ○ Keep floating
○ Keep as part of
● Keep ass. from: A-CLIENT
Understanding: the object which invokes a command object. Can be reached from the client.

Figure 3: *A snapshot of some of the DYNAMO tools.*

model as well as a tool that allows editing the structure of the individual scenarios. The *message browser* allows editing of the details of a single message, such as parameters, pre- and post-situations, message result, and the message understanding. The *object browser* allows editing of the details of an object (name, class, object keeping, and the object understanding).

In addition, we support an *interaction diagram editor*, which allows the designer to view and edit a single scenario as an interaction diagram. An example of a DYNAMO interaction diagram has been shown in figure 2. One of the very useful aspects of the DYNAMO interaction diagrams is the indication of object life times along the vertical lines that represent each individual object. The object life time information in an interaction diagram shows the scene of objects graphically, at any time of a scenario. We also explicitly present the nesting of messages to a particular object by sliding the messages a few pixels to the right of the line which represents the object in the diagram. Furthermore, the interaction diagram allows creation of new messages and editing of existing messages by direct manipulation of the arrows in the diagram.

The browsers mentioned above can be thought of as syntax-directed structure editors of a dynamic model. The designer's interface to these tools consist of a number of fields in which information can be entered, a number of lists which may be extended and pruned, and a variety of command buttons, via which the browsers can be controlled. In contrast, the interaction diagram editor provides a graphical user interface, which supports one particular diagrammatic view of a dynamic model.

Figure 3 shows a snapshot of the dynamic model browser, the interaction diagram and the object browser on the example discussed in section 3.2.

While working with the objects and object interactions of a dynamic model, the associated static model may, in several respects, serve as an abstraction of the dynamic model. The reason is that a number of objects are abstracted to the class of which they are instances. Similarly, a number of messages are abstracted to the method that is invoked when the message is sent. Consequently, it is useful to consult the associated static model in order to get an *overview* of the dynamic model. The *static model browser* of DYNAMO presents such an overview to the designer.

The static model browser enumerates the classes that have instances in the dynamic model. When selecting a class, the list of messages sent to objects of this class is shown. These are interpreted as methods in the classes. It is possible to generate class descriptions in various concrete syntaxes, either as interface descriptions or as outlines of full classes. The visibility of methods (private or public) is also estimated. Method bodies are generated by merging the relevant scenarios. If more than one scenario is given for a particular message, this may cause the generation of a selection (an if-then-else) in the body of a method. The assertions (start and end situations, as described in section 3) play an important role here. If part of a scenario is repeated later on in the scenario this may cause us to conjecture a repetition (a loop). For further details about this please consult [Nørmark, 1997a].

Class relations (aggregations and associations) can be derived from the similar object relations. We do not, however, plan to infer class inheritance relations from the dynamic model. Although one could consider sets and subsets of messages to objects as the basis for deriving some inheritance relations, we do not think that automatically derived inheritance information will be useful for modelling purposes.

5 Discussion

In this section we first discuss the differences and the similarities between programs and scenarios. Following that we outline the modifications and extensions to the abstract dynamic modelling language we plan in the future.

5.1 Scenarios versus programs

A program (relative to some object-oriented programming language) is a prescription of the object dynamics that we model. The program itself represents a static model, in which a number of structures among classes are emphasized. However, the program is also interpreted and thereby executed. During execution the program prescribes the *dynamic development* among the objects, possibly as a function of some input data.

A scenario is a *direct* description of the dynamic development at a high level of abstraction. The program is a more *indirect* description of the dynamic development, because most of the description is directed towards classes, as opposed to objects. A set of scenarios represent examples of dynamic development whereas the program is a more general and more complete description, which ideally takes all necessary cases into consideration. However, by providing a number of different scenarios of a single, top-level message, we may cover the essential cases, both normal and abnormal.

As already discussed earlier in this paper, we carefully protect the abstraction barriers in an object-oriented program. One good reason for this stems from maintainability. During the design phases, however, we deal with interaction among objects that cross the abstraction barriers. This causes no harm at this level. Quite the opposite, it gives us a chance to understand some important overall patterns of interaction among objects that are part of our design.

Execution is not meaningful on scenarios. Scenarios may, however, be *explored* and *analyzed* by the designer. The purpose of such exploration and analysis is many-fold:

1. To understand and develop adequate and desired interaction patterns among objects as directly as possible.
2. To derive aspects of a static model from the dynamic model, for instance class interfaces and method outlines.
3. To provide an overview of aspects of the dynamic model, for instance the exact set of objects on the scene at a given point in the scenario.
4. To check for possible inconsistencies in the dynamic model, for instance use of unprovided objects.

The exploration of a dynamic model will typically lead to a redesign and an "improvement" of the model, for instance a re-distribution of responsibilities among the objects.

5.2 Planned language extensions

The dynamic modelling language described in section 3 of this paper has not yet found its final form. Here we discuss a number of issues that are likely to affect the next version of the language.

Our current model is relatively weak with respect to description of *object state*. Our main philosophy has been to describe object state informally. In the future we are likely to introduce a concept of *named states* in the modelling language. As a consequence of sending a message from one object to another, the receiving object may turn into another state. By introducing named states in scenarios, we will be able—in a straightforward way—to generate some elements of state machines directly from the scenarios. The reason is that the information about states and state transitions is contained directly in the dynamic model. Notice that this approach is different from (and simpler than) the approach taken in [Koskimies and Mäkinen, 1994], where state machines are synthesized from scenarios without any integrated state concept.

In the current version of the language we support the part-of and associated-from relations between objects via object keepings, but these are static relations. We are likely to extend this aspect so that a more versatile set of object relations can be supported, and so that these relations become more dynamic (in the sense that the relations can be changed during a scenario). In the ideal situation we hope to be able to integrate named states and dynamic relations to one overall *object state concept.* The object state concept is likely to be part of a more formalized assertion sub-language of DYNAMO.

In the current version of the abstract language, there are no means for a scenario S to "call" another scenario T. It is currently necessary to inline T into S. This simplifies matters because there is no need for any actual-to-formal parameter passings. In fact, there is currently no notion of formal parameters to operations in DYNAMO. One obvious drawback of this approach is the need for duplication of T into a number of scenarios. This leads to substantial maintenance problems of a design.

We do not support any kind of selection (if-then-else) or iteration (loops) in the dynamic modelling language of DYNAMO. If variations are needed in a scenario we have to make two or more scenarios, one for each variation. The assertions distinguish between the variations. If iteration is needed, we can only exemplify this (say with two or more repetitions in a scenario). Recall, however, that we try to infer both selections and iterations from DYNAMO scenarios. We have no plans to introduce control structures in the dynamic modelling language along the lines of algorithmic scenarios found in [Koskimies *et al.*, 1996a]. We think that one of the strengths of the current scenario language is it simplicity, and we want to keep it this way also in the next version of the language.

Object provision is a central concept in the current version of the language. Object provision is namely a mechanism to state the relevance of a new or existing object, with certain initial properties (object state and relations to other objects on the scene). In addition, we are forced to use the object provision mechanism to access an object Q which is somehow related to a named object N in the current scenario.[2] The reason is that the language, as of now, lacks dereferencing mechanisms and mechanisms for "object relation navigation".

6 Similar work

As already noticed in section 2 our work with scenario-based dynamic modelling is related to use case driven design, as described by Jacobson et al. in [Jacobson *et al.*, 1992]. A set of scenarios may be seen as a formal elaboration of a use case.

The scenario concept is important in many object-oriented design notations and methods, such as in UML [Rational Software Coorporation, 1997] and in earlier work by Booch, Rumbaugh, and Jacobson [Booch, 1994, Rumbaugh *et al.*, 1991, Jacobson *et al.*, 1992]. However, the main emphasis in these authors' work is centered around diagrammatic notations of scenarios. UML-like sequence diagrams emphasize the temporal order of the object's interactions. UML-like collaboration diagrams show the object interactions in terms of a graph, where the vertices represent objects and the edges messages and links between objects. Chapters 5 trough 8 of the version 1.0 UML Notation Guide [Rational Software Coorporation, 1997] contain some good overview of use cases and various notations of scenarios.

In the book "Seamless Object-oriented Software Architecture – Analysis and Design of reliable Systems" [Walden and Nerson, 1995] Nerson and Waldén devote a chapter to the description of dynamic models in the BON notation. They first describe a number of "event charts" and other charts (which are just tables with selected relations among elements in a dynamic model). They reject state transition diagrams and finite state machines as the only reasonable formalism for description of dynamic models because of the mismatch between static OO models and state transition diagrams. Following this they develop a scenario concept and introduce some simple diagrams, through which a number of objects and 'message relations' (in terms of arrows) are drawn between objects. The temporal aspects are captured by numbers on the message relations.

Koskimies et al. have explored the interplay between scenarios and state machines [Koskimies *et al.*, 1996a, Koskimies *et al.*, 1996b, Koskimies *et al.*, 1994]. The most important result of this work is an algorithm for automatic generation of state machines from a set of scenarios [Koskimies and Mäkinen, 1994]. The algorithm is based on a relatively old algorithm by Biermann et al. which deals with program construction from examples [Biermann and Krishnaswamy, 1976]. The results of the

[2]The underlying problem is that the object Q originally was provided with a unique name, but that name is not in the scope of the current scenario. However, the object has survived because it stands in a relation to an object which is directly on the scene of the current scenario.

work by Koskimies et al. has been implemented in the SCED tool. SCED is considered a CASE tool that allows for editing both scenarios and state diagrams. The more interesting capabilities of SCED are, however, the possibilities of integrating the work with the two notations.

Salmela et al. identify the need for graphical animation techniques for object-oriented modelling purposes [Salmela *et al.*, 1994]. Their main observation is the same as the observation behind DYNAMO: it is very difficult to capture a "living" dynamic model in a static medium. Salmela et al. take the full consequence of this observation by proposing a visualization schema for dynamic object-oriented models based on animation. Their work is especially directed towards real-time software.

As part of the Beta Mjølner programming environment, Christensen and Sandvand describe a CASE tool called Freja [Christenen and Sandvad, 1996]. One of the main ideas behind Freja is to provide an integrated support of design models and programs via a shared, internal representation. Consequently, a static design diagram may be created as a *view* of an underlying program representation. In addition, Freja supports the creation of a kind of collaboration diagram. In Freja, these diagrams stem from a static analysis of a Beta program.

The Scene system [Koskimies and Mossenbock, 1996] is able to produce scenario diagrams from program executions. As such, the Scene system follows a different approach from Freja (described above). The main focus in the Scene work is, however, to use scenarios for understanding and browsing existing software. As such, there are only few similarities between Scene and the DYNAMO work.

Scenarios are also used in the human-computer-interaction area for analyzing user tasks [Carroll, 1994, Rosson and Carroll, 1995]. In this setting the scenario concept tend to be narrative descriptions of the user requirements, problems, and observations. Although informal, the scenarios used here are organized systematically in order for the designer to bring order to concrete observations of the interaction between the user and the computer. In [Rosson and Carroll, 1995] a set of tools are described, the intention of which is to narrow the gap between task analysis (oriented towards the user's interaction with a computer system) and object-oriented analysis and design (oriented towards the organization of problem-oriented concepts and the structure of the object-oriented software). The paper cited above contains discussions on a number of issues (scenario evolution, completeness, and consistency) which turn out to be relevant for both design of user tasks and design of software using a scenario-based approach.

The original and early ideas behind DYNAMO are presented in a technical report [Nørmark, 1996a], in which we perform a detailed comparison with Booch's, Rumbaughs's, and Nerson's diagrammatic notations, especially with respect to the expressive power when new objects are introduced during a scenario. The ideas behind the early DYNAMO tools are described in [Nørmark, 1996b]. The DYNAMO home page on the Internet [Nørmark, 1997c] provides full information on our past and present work, including a description of the tools which we use to create and explore dynamic models.

Acknowledgements. I would like to thank Mike Manthey for help during the process of writing this paper.

This research was supported by the Danish Natural Science Research Council, grant no. 9400911.

References

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

[Booch, 1994] Grady Booch. *Object-oriented analysis and design with applications, second edition.* The Benjamin/Cummings Publishing Company Inc., 1994.

[Carroll, 1994] John M. Carroll. Making USE a design representation. *Communications of the ACM*, 37(12):29–35, December 1994.

[Christenen and Sandvad, 1996] Michael Christenen and Elmer Sandvad. Integrated tool support for design and implementation. In Lars Bendix, Kurt Nørmark, and Kasper Østerbye, editors, *Proceedings of the Nordic Workshop on Programming Environment Research, NWPER'96*, pages 169–182. Department Computer Science, Institute for Electronic Systems, Aalborg University, R-96-2019, May 1996 1996. http://www.cs.auc.dk/∼normark/-NWPER96/proceedings/proceedings.html.

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

[Graham, 1996] Ian Graham. Task scripts, use cases and scenarios in object oriented analysis. *Object Oriented Systems*, 3:123–142, 1996.

[Jacobson *et al.*, 1992] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley Publishing Company and ACM Press, 1992.

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

[Koskimies and Mossenbock, 1996] K. Koskimies and H. Mossenbock. Scene: Using scenario diagrams and active test for illustrating object-oriented programs. In *Proceedings of the 18th international conference on software engineering, Berlin, Germany, March 1996.*, March 1996.

[Koskimies *et al.*, 1994] Kai Koskimies, Tatu Männistä, Tarja Systä, and Jyrki Tuomi. SCED - an environment for dynamic modeling in object-oriented software construction. In Boris Magnusson et al., editor, *Proceedings of the Nordic Workshop on Programming Environment Research, NWPER'94, Lund*, pages 217–230, 1994.

[Koskimies *et al.*, 1996a] 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.

[Koskimies *et al.*, 1996b] Kai Koskimies, Tatu Männistä, Tarja Systä, and Jyrki Tuomi. On the role of scenarios in object-oriented software design. In Lars Bendix, Kurt Nørmark, and Kasper Østerbye, editors, *Proceedings of the Nordic Workshop on Programming Environment Research, NWPER'96*, pages 53–69. Department Computer Science, Institute for Electronic Systems, Aalborg University, R-96-2019, May 1996 1996. http://www.cs.auc.dk/∼normark/-NWPER96/proceedings/proceedings.html.

[Meyer, 1988] Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, 1988.

[Nørmark, 1987] Kurt Nørmark. *Transformations and Abstract Presentations in a Language Development Environment.* PhD thesis, The Computer Science Department, Aarhus University, Denmark, February 1987. DAIMI PB-222.

[Nørmark, 1996a] Kurt Nørmark. Dynamic models in object-oriented design. Technical Report R-96-2005, Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, February 1996.

[Nørmark, 1996b] Kurt Nørmark. Tools for exploration of dynamic models in object-oriented design. In Lars Bendix, Kurt Nørmark, and Kasper Østerbye, editors, *Proceedings of the Nordic Workshop on Programming Environment Research, NWPER'96*, pages 185–195. Department Computer Science, Institute for Electronic Systems, Aalborg University, R-96-2019, May 1996 1996. http://www.cs.auc.dk/∼normark/-NWPER96/proceedings/proceedings.html.

[Nørmark, 1997a] 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.

[Nørmark, 1997b] Kurt Nørmark. Dynamo: A set of tools for dynamic modelling. Forthcoming paper, 1997. Preliminary version available on http://www.cs.auc.dk/∼normark/dynamo.html.

[Nørmark, 1997c]
Kurt Nørmark. The WWW home page of the DYNAMO project. http://www.cs.auc.dk/∼-normark/dynamo.html, 1997.

[Rational Software Coorporation, 1995]
Rational Software Coorporation. Unified method - notation summary 0.8. Available from http://www.rational.com, 1995.

[Rational Software Coorporation, 1997]
Rational Software Coorporation. Unified method - notation guide 1.0. Available from http://www.rational.com, 1997.

[Rosson and Carroll, 1995] Marry Beth Rosson and Johm M. Carroll. Integrating task and software development for object-oriented design. In *CHI'95: Human Factors in Computing Systems*. ACM, 1995.

[Rumbaugh *et al.*, 1991]
James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall International, 1991.

[Salmela *et al.*, 1994]
Marko Salmela, Marko Heikkinen, Petri Pulli, and Reijo Savola. A visualisation schema for dynamic object-oriented models of real-time software. In Boris Magnusson et al., editor, *Proceedings of the Nordic Workshop on Programming Environment Research, NWPER'94, Lund*, pages 73–86, 1994.

[Walden and Nerson, 1995] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice Hall, 1995.