DYNAMO
A set of Tools for Dynamic Modelling

Kurt Nørmark
Aalborg University
Denmark[*]

June 26, 1997

Abstract

The DYNAMO work is about modelling in terms of objects, object relations, and object interaction. Our work contributes to the branch of object-oriented design called dynamic modelling. Our basic approach is to develop scenarios early in the design process, and to automatically derive information about classes, class relations, and methods from the scenarios. A scenario is a concrete example of interaction among objects. The objects may be pre-existing, or they may be provided as integral parts of the scenario. As an important part of our work we have implemented an environment that allows us to build and explore dynamic models. In this paper we will describe our experience with the development of the tools in the DYNAMO environment. The main insight in the paper is about the internal model representation (in terms of abstract syntax trees) in relation to the external views (in term of textual presentations as well graphical interaction diagrams).

1 Introduction

In this paper we will describe and discuss the DYNAMO environment, which consists of a set of tools for construction and exploration of dynamic object-oriented models. DYNAMO is implemented in Common Lisp and CLOS [7] on a PC platform, using Allegro Common Lisp for Windows.

In DYNAMO the designer is able to specify examples of interaction among a number of passive objects. The objects can either be pre-existing, or they can appear during the course of the interaction. By specifying the concrete interplay between the objects a number of significant elements of the static design are

[*]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.

specified as well. One of the promises of DYNAMO is to synthesize and extract this information. Some aspects of a specification are formalized in DYNAMO. Besides the messages and their parameters, the life times of objects, the classes of objects, the relations among objects, and the result of function-like messages are formalized in the current version of DYNAMO. But we also support and rely on informal descriptions of the interactions and the objects involved.

The work with DYNAMO is oriented towards three different research areas:

1. Definition of a dynamic modelling language for object-oriented design.
2. Automatic generation of static models from dynamic models.
3. Construction of a set of tools for model building and exploration.

The modelling language is defined at an abstract level. Consequently, a dynamic model is represented as an abstract syntax tree derived from an abstract grammar. The current version of the modelling language is described in details in a separate paper [16].

It is one of the basic ideas of the DYNAMO work that dynamic models should be made early in the design phase, and before any static model is elaborated. The idea of an early deployment of dynamic modelling is based on the hypothesis that designers think in terms of objects rather than classes in the creative phase of the design process. We attempt to generate substantial parts of the static models from the dynamic models. A detailed account of this part of the work can be found in a separate paper [15].

The third research area is covered in the present paper. We start with a section summarizing the basic concepts of the modelling language. Following that we discuss the language implementation approach, which is centered around a custom made structure editor with a clean internal representation, and we contrast our approach with other possible implementation approaches. We also describe the evolution of the environment from a very simple system to the environment, as it exists today. In the main section of the paper we describe a particular view of a model, called an interaction diagram. Interaction diagrams are integral parts of the notation in many OOD methods [2, 18, 6], and they are important in several case tools. We will argue that interaction diagrams are very suitable views on a dynamic model, both in an absolute sense and relative to other possible views. In addition, we will describe some of the detailed elements of our diagrammatic notation together with some direct manipulation interaction techniques, we have developed in the DYNAMO project.

2 The basic DYNAMO concepts

The most important concept in DYNAMO is scenarios. A *scenario* is a concrete example of interaction among a number of objects. A scenario is described in terms of a message from the "surround" to a receiver object. The receiver, may in turn, send a number of messages to other objects, and so on recursively. Thus, a scenario is as a tree of messages, and as always when dealing with trees, it is possible to identify the tree (the scenario) with the root node of the tree (the top-level message). In this paper we will call this structure for a *message tree*.

A *message* m is characterized by:

- The receiver object.
- The actual parameters.
- The situation that is present just after the parameters have been passed to m (see below).
- An understanding that explains m in general (see below).
- An understanding of the particular instance of the message.
- The objects provided by m (see below).
- The messages sent from the receiver of m to other objects.
- The result of sending the message.
- The situation that is present just before m is completed, and is about to return the control to the sender.

The pre and post-situations are assertions that describe the situation at some given points of time during a scenario. The assertions are *situational* in contrast to *prerequisitional*. It means that a pre-situations and post-situations describe the situation per se, at the point where they are located. A prerequisitional assertion expresses a requirement to the state of the computation. Prerequisitional assertions are known from the pre-conditions of the programming language Eiffel [13].

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, or 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 object provision time. In these situations, object provision is identical to object creation. Either way, 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 unique names 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 maintain the designer's intuition about the messages and objects.

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-based 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 more holistic picture of the way we intend to solve a problem.

We are currently in the process of changing the modelling language of DYNAMO. The major changes are oriented towards a formalization of the object state concept (including named states and arbitrary, binary relations among objects), a possibility of applying existing scenarios from other scenarios (including a general abstraction mechanisms on scenarios) and a refinement of the object provision concept (in which we distinguish between object creations, pure object provisions, and object re-provisions).

3 The language implementation approach and its evolution

As mentioned in the introduction, the dynamic modelling language is syntactically defined at an abstract level. This is attractive because it gives us a conceptually clean internal representation of a model in terms of an abstract syntax tree. All the tools described in this paper operate on syntax trees. As an additional advantage, we are not forced into premature decisions about external presentations of the models. Consequently, we do not need to speculate on the traditional issues such as concrete syntax and parsing. More important, perhaps, our work can—in the starting point—be concentrated around the important underlying concepts instead of a more superficial diagrammatic notation.

Given that our primary, internal representation is an abstract syntax tree we need some editing tools via which to construct such a data structure. The general solution is to create a structure editor the commands of which directly manipulate the internal representation. It is possible to derive such editors automatically from an abstract grammar, in which case we talk about syntax-directed editing [20, 14, 19] and editor generators [17]. However, in this project we hand crafted the DYNAMO structure editor. This has been a reasonable approach because the modelling language is small, and because we wanted to experiment with a number of "special user interfaces" instead of "standard syntax-directed editor interfaces".

At the outset we wrote an abstract grammar for the dynamic modelling language and we hand coded the grammar as a set of classes in CLOS. The abstract grammar is shown in appendix A. We used the following two grammar structuring principles [3, 12] when writing the classes:

1. *Alternatives are implemented as specialization*:
   Given n alternative productions

   A → A1 | A2 | ... | An

   where A as well as Ai, i = 1..n, are nonterminals, we implement the CLOS class

   `(defclass Ai (A))`, for i = 1...n.

   Thus, `Ai` is a trivial class which inherits from `A`. We may put one or more attributes into a class `Ai`, in case the possible specializations of `Ai` all share a given syntactic constituent. (This is a way of factoring out similar parts of closely related concepts).

2. *Constructors are implemented as aggregation*:
   Given a "constructor production"

   B → B1 B2 ... Bm

   we implement the class

```
(defclass B (A)
  ((B1
     :accessor B1
     :initarg :B1)
   (B2
     :accessor B2
     :initarg :B2)
   ...
   (Bm
     :accessor Bm
     :initarg :Bm)))
```

   The class `A` is one of the alternatives derived from the principle described above, or (if no such alternative exists) a common superclass of all syntactic categories of the modelling language.

We see that an instance of a constructor holds instances of sub-constructs. Furthermore we see that alternatives do not add to the depth of the syntax trees. In other words we avoid adding depth to the syntax tree due to application of traditional single productions [1].

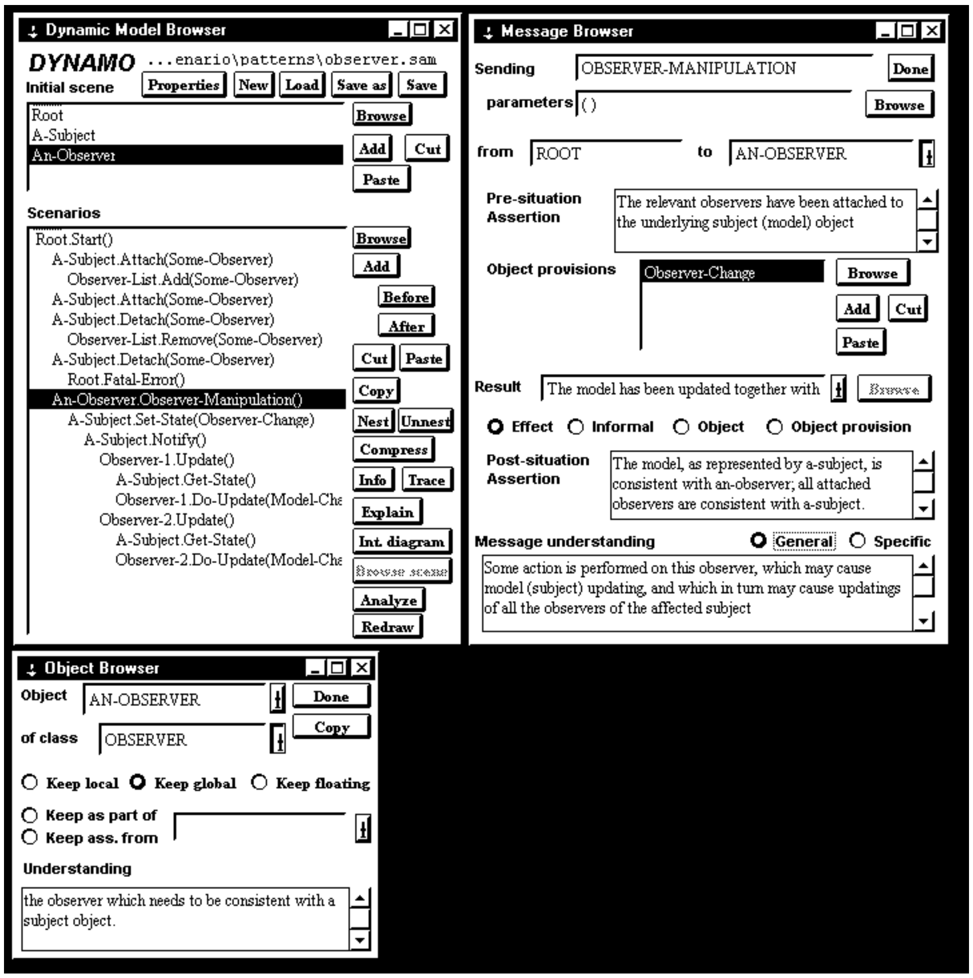
Given the classes we are, in principle, able to make dynamic models by instantiating the classes that represent the modelling concepts. In practice, however, need also some *external presentation* of the abstract syntax trees and a *persistent file representation* of the model. Furthermore we need some *set of editing commands* that allows us to modify an existing model.

In the very first version of the tool we made a repertoire of tersely named Lisp functions via which we were able to instantiate and insert the various parts of a dynamic model. The functions prompted the designer for the terminal aspects of a model. We used the notion of a *current focus*, which can be moved around in the syntax tree, and which serves as an implicit parameter to most of the editing commands. The external presentation was realized by a pretty printer based on a straightforward tree traversal during which the current focus was indicated clearly. The persistent representation of a model consist of nested activations of Lisp constructor functions (along the style recommended by Keene [8], page 163). Roughly, there exists a constructor function for each constructor found in the abstract grammar. Using these simple means we were able to construct our first dynamic models, and to get some important preliminary experience with our ideas. The investment was only a few days of programming efforts.

As a contrast to our approach, it would have been possible to follow a more conventional language implementation technique. Following such an approach we should define the concrete syntax of the modelling language, and write models in a text editor. Subsequently a model should be parsed and processed appropriately. Using state-of-the art parser generators this is a straightforward effort too. However, we loose the ability to manipulate the model representation directly, and perhaps more important, we will be likely to manipulate all the model details simultaneously, as a monolithic mass of information. As another possibility, we could have focussed on a given diagram notation in the starting point, and thus implement "a drawing tool" with early emphasis on graphical means of expression. In our opinion this not attractive either, because evidence from notations in contemporary OOD methods indicate that the superficial graphical details of the notation tend to attract more interest than the underlying concepts of the model.

One of the severe weaknesses of the simple approach described above turned out to be the lack of editing flexibility, especially in situations where we had to make a lot of small changes to a model. Due to a relatively coarse-grained repertoire of editing commands such changes were difficult and awkward to carry out. As a consequence we added a number of browsers to DYNAMO.

At top level, the *dynamic model browser* allows us to manipulate the initial scene and the set of scenarios of the model. As a simple convention, the top-level



DYNAMO ...enario\patterns\observer.sam

Initial scene

```
Root
A-Subject
An-Observer
```

Scenarios

```
Root.Start()
  A-Subject.Attach(Some-Observer)
    Observer-List.Add(Some-Observer)
  A-Subject.Attach(Some-Observer)
  A-Subject.Detach(Some-Observer)
    Observer-List.Remove(Some-Observer)
  A-Subject.Detach(Some-Observer)
    Root.Fatal-Error()
  An-Observer.Observer-Manipulation()
    A-Subject.Set-State(Observer-Change)
      A-Subject.Notify()
        Observer-1.Update()
          A-Subject.Get-State()
          Observer-1.Do-Update(Model-Cha
        Observer-2.Update()
          A-Subject.Get-State()
          Observer-2.Do-Update(Model-Cha
```

Sending: OBSERVER-MANIPULATION

parameters: ()

from: ROOT to: AN-OBSERVER

Pre-situation Assertion: The relevant observers have been attached to the underlying subject (model) object

Object provisions: Observer-Change

Result: The model has been updated together with

[✓ Effect] Informal Object Object provision

Post-situation Assertion: The model, as represented by a-subject, is consistent with an-observer; all attached observers are consistent with a-subject.

Message understanding: [✓ General] Specific

Some action is performed on this observer, which may cause model (subject) updating, and which in turn may cause updatings of all the observers of the affected subject

Object: AN-OBSERVER

of class: OBSERVER

Keep local [✓ Keep global] Keep floating

Keep as part of

Keep ass. from

Understanding: the observer which needs to be consistent with a subject object.

Figure 1: *Examples of the DYNAMO browsers.*

scenario is a pseudo scenario called start, the sub-scenarios of which make up the list of top-level scenarios of the dynamic model. The dynamic model browser makes it, in addition, possible to manipulate the message structure of each of the scenarios. (However, in the most recent version of the environment, this functionality is taken over by the interaction diagram editor to be described in section 4 below). The *message browser* allows the designer to edit the details of a single message: the message name, the receiver, the parameters, the pre- and post situations, the object provision of the message, the result of the message, and the specific and the general understandings. Similarly, the *object browser* allows editing of the details of an object provision: the object name, its class, the object keeping, and the object understanding. Figure 1 shows an example of the three browsers.

The browers have all been implemented by means of the interaction builder of the Lisp system. As such, the user interface elements of the browsers are drawn by direct manipulation and attached to suitable Lisp functions which are activated when events occur on the browsers. With the browsing interface it became possible to change minor details of a model, (such as the second character of the third parameter of a message) in a very convenient manner.

In the next section we will discuss a tool that allows the designer to edit a scenario as a diagram in which the interactions are illustrated graphically instead of textually. As mentioned above, this reduces the dynamic model browser to a tool which keeps track of the overall elements of a model: the intial scene and the set of scenarios.

4 The interaction diagram editor

Despite the flexibility of the browsers described above, it turned out to be very difficult to communicate the individual scenarios from one designer to another via textual means only. One reason is the notational habits in the area, which undoubtedly are oriented towards diagrammatic notation. Another related reason is the "fact" that diagrams are superior to text when the problem is to present a general overview at a high level of abstraction.

As a consequence of these observations it was decided to implement one of the rather well-known OOD diagrammatic notations in DYNAMO. The diagrams are implemented on top of the internal AST representation of a model, and as a supplement to the browsers.

4.1 Object graphs and Interaction Diagrams

There are basically two different diagramming techniques which have to be considereded as a presentation of scenarios. The first is an *object graph* in which the nodes represent objects, and the links represent messages (or relations between objects via which it is possible to send messages). In the UML notation (version 1) object graphs are known as collaboration diagrams. The second is an *interaction diagram* in which the objects are presented as vertical lines, and messages are represented as arrows between the objects. Interaction diagrams are called sequence diagrams in UML. Figure 2 shows comparable and stylized examples of the two kinds of diagrams. We will now discuss these two kinds of diagrams as the basis for a tool in DYNAMO.

An object graph is probably the most intuitive notation. People who discuss an object-oriented design at a blackboard tend to use the object graph notation. As pointed out in [22], the designer is free to place an object anywhere in the two-dimensional space. This allows for natural grouping of related objects. As a contrast to interaction diagrams, an object diagram is symmetric in its use of the two dimensions of the drawing surface. However, this can be regarded as a disadvantage from the tool perspective, because it is inherently difficult to

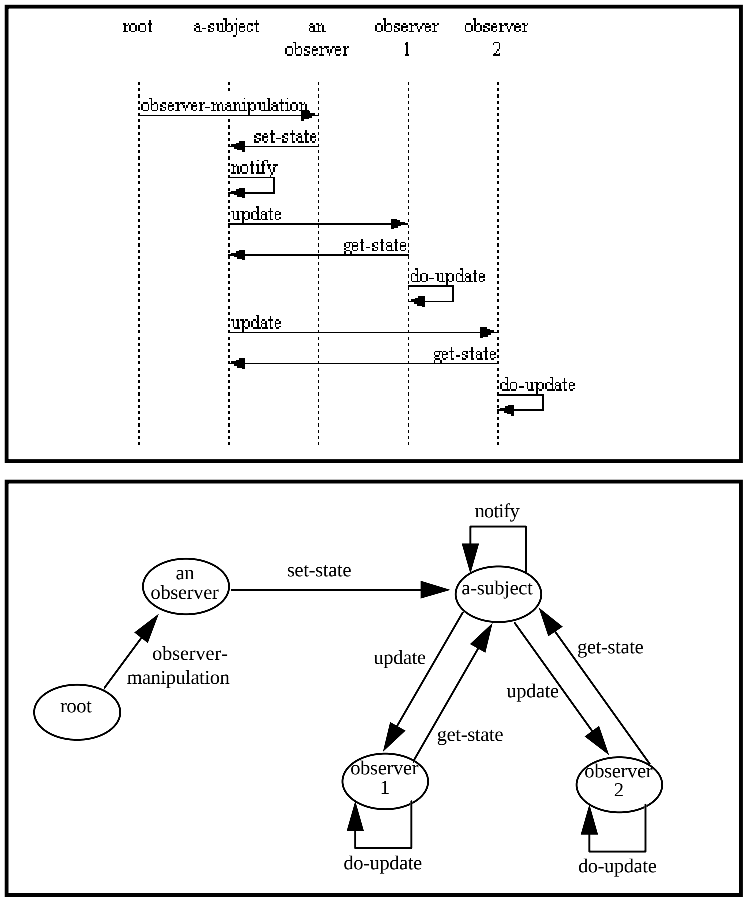


Figure 2: *An interaction diagram (above) and the similar object graph (below).*

make automatic layout of graphs in a two dimensional surface [5].

It is possible to interpret an interaction diagram as an object graph too. Following this interpretation the nodes of the graph are rendered as vertical lines, and the edges are horizontal arrows, or bended arrows from one node to itself. The node layout is linear and therefore trivial. The only special features of an interaction diagram, regarded as an object graph, is the significance attached to the relative vertical positions of the edges. The vertical dimension represents time. If timing aspects have to be shown in a more conventional object graph it requires additional means, such as sequence numbers on messages.

From a tool perspective it is relatively straightforward to draw an interaction diagram, primarily because there no complicated layout problems that have to solved. However, as pointed out in [22], the number of objects that can be shown on the screen at a given point of time is lower than the number of objects that can be scattered around in the two dimensional plane of an object graph.

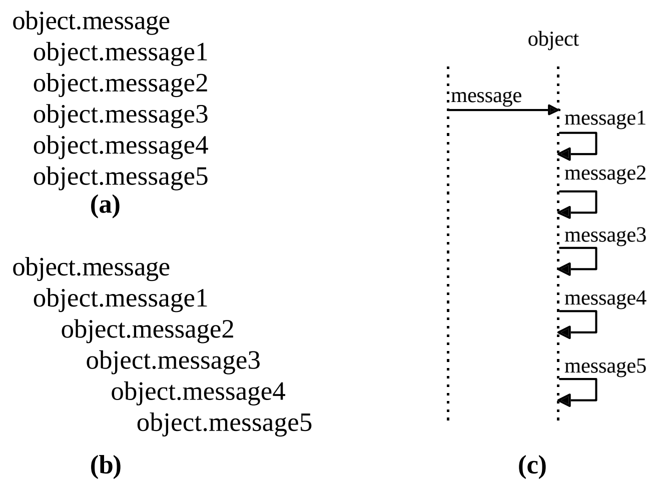


Figure 3: *Two different message trees with identical interaction diagrams.*

As it appears from this discussion there are relative strengths and weaknesses in both notations. We chose the interaction diagram for DYNAMO. We will now address a number of notational and interactional issues in the elaboration of the interaction diagram as a tool in DYNAMO.

4.2 The graphical notation in interaction diagrams.

The primary internal DYNAMO representation of a scenario is the message tree (see section 2), in which a node N is a message to a given object O. Sons of N represent sub-messages from O to other objects (or to O itself), and so on recursively. The dynamic message browser presents a message tree in a simple and straightforward way, as nested lists with textual indentation. (This can be seen in figure 1). The structure of an interaction diagram is not entirely homomorphic with a message tree. The basic, structural difference between the two is that a given object only appears once in an interaction diagram, but may appear several times, at several levels in a message tree (as names representing the object). As an implication of this an interaction diagram may be ambiguous relative to the underlying message tree. Figure 3 shows an extreme example of two different message trees which cannot be distinguished in the usual rendering of an interaction diagram.

It seems to be important to preserve the levels of messages (message, sub-messages, sub-sub-messages, etc.) in an interaction diagram. If not done, the interaction diagram may become ambiguous in a number of ways. Several authors (e.g., Booch [2]) use the "box notation" in figure 5 for message activations. However, it is difficult to generalize this particular notation to deep nesting, as present in figure 3(b), for instance. As a consequence, we came up with a new

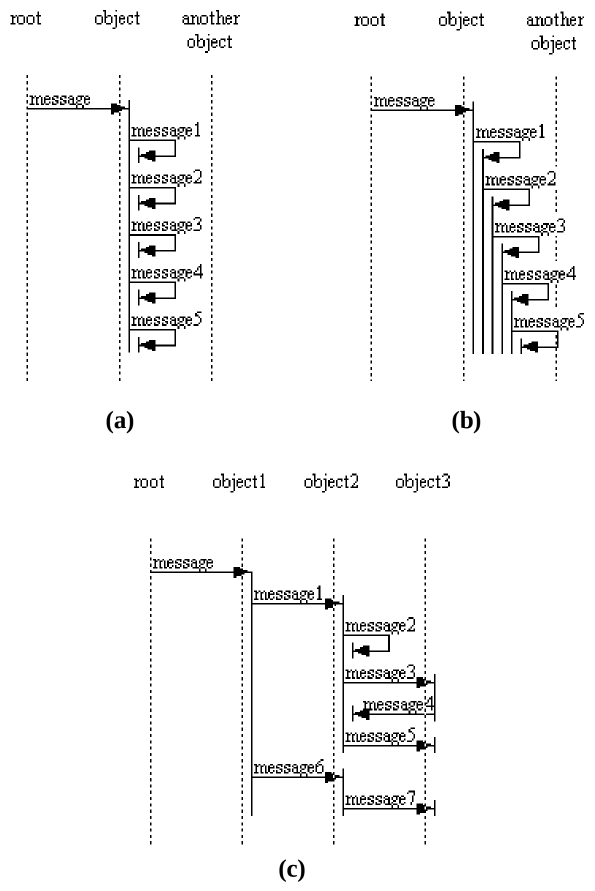


Figure 4: *DYNAMO interaction diagrams of message tree* **a** *and* **b** *from figure 3 and a diagram similar to the "Booch style diagram" of figure 5*

notation in the DYNAMO project, in which each message activation is shown as an *activation line* parallel with the object line, but with a few pixels offset to the right. The principle is that each activation of a message M on an object O gives rise to a new activation line on O. In more graphical terms, each arrow head points at a new activation line. In a design without multiple, active objects (the present limitation of DYNAMO) a message does not return before the returns of all sub-messages. In such situations it is always the case that inner activation lines of O are temporally contained in outer activation lines of O.

Interaction diagrams with activation lines show the message tree of the underlying scenario in an unambiguous way. In addition, it is easy to grasp the stack of activations by following the edges from object to object. Figure 4 illustrates DYNAMO interaction diagrams for the two message trees in figure 3(a) and (b), and a diagram similar the one in figure 5. The interaction diagrams in figure 4 are taken directly from the DYNAMO environment. As it is illustrated in the figure, the adopted notation can show a message trees of "reasonable depths" as interaction diagrams without running into graphical problems. (In a scenario-based design approach it is difficult to imagine situations where more than five simultaneous activations exists on a given object. If a need for more levels of activations should arise, it is sufficient to adjust the mutual distance between the object lines of the diagram).

Objects can appear and disappear during a scenario. Object appearance may in some situations be equivalent to object creation, but in others it may signal that an object becomes relevant for the scenario we are specifying. In the DYNAMO modelling language it is possible to specify when an object becomes relevant via the object provision mechanism. Furthermore, the period of relevance can be specified via the object keeping concept (see section 2). It is important to reflect graphically the object provision time and the period of object relevance. In DYNAMO interaction diagrams each object is presented as a dashed object line, which is "open" in the positive time direction. We superimpose a closed, colored line segment on the object line which shows the period of time the object is relevant for the purposes of the scenario. We call this line segment for an *object relevance-line.* It should be noticed that the object relevance line is contained in the *object life-line* (the time span between object creation and object destruction), as defined in UML. Figure 6 shows an example of an interaction diagram with object relevance lines.

The color of the object relevance line signals the "place of appearance" of the object relative to the scenario structure. Via use of colors we distinguish between objects provided globally (for instance on the initial scene), object provided in parameter lists, object provided in the body of a message, and object provided as part of the message result.

It is, of course, possible to add additional graphical details to an interaction

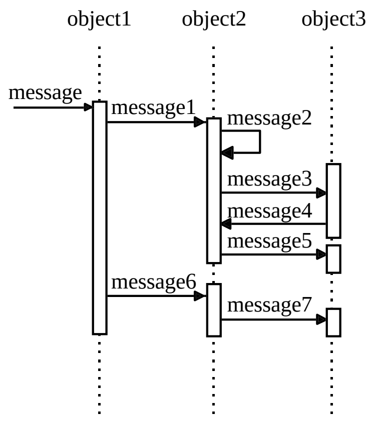


Figure 5: *An interaction diagram in "Booch style".*

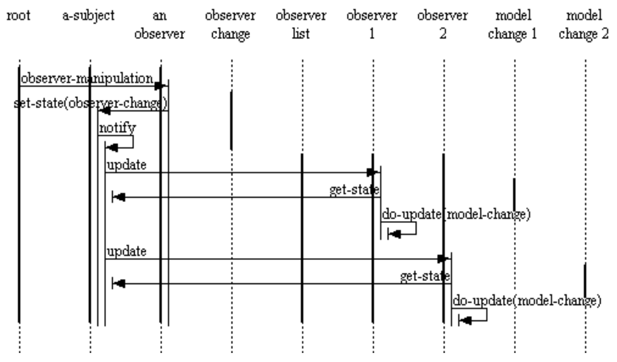
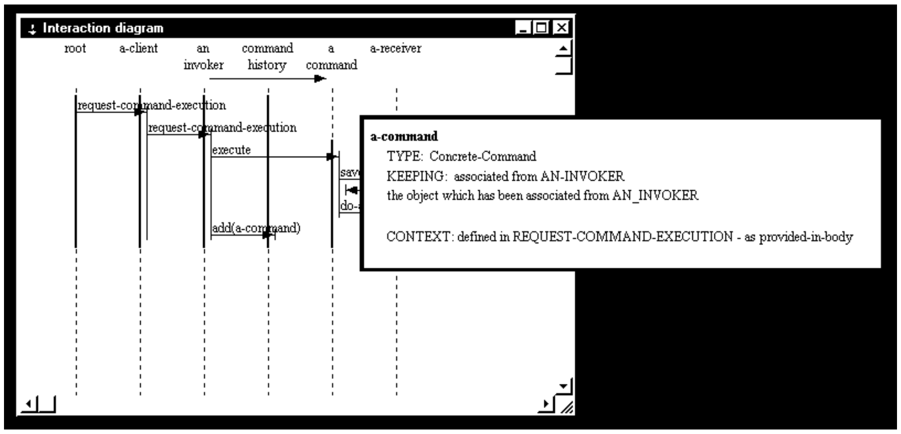


Figure 6: *An interaction diagram for the observer design pattern with object relevance lines (in black and white only). The observer design pattern was shown in figure 1 in a Dynamic Model Browser.*



a-command
TYPE: Concrete-Command
KEEPING: associated from AN-INVOKER
the object which has been associated from AN_INVOKER

CONTEXT: defined in REQUEST-COMMAND-EXECUTION - as provided-in-body

Figure 7: *An example of a pop-up box explaining details about the object* `a-command`*, and a pop-up arrow that shows the association between* `an-invoker` *and* `a-command`*.*

diagram, hereby making it possible to illustrate more and more aspect of the dynamic model through the interaction diagram. However, the more details we add, the more difficult it becomes to get the general overview of the message structure. The UML sequence diagrams (and collaboration diagrams) contain several examples of fairly detailed graphical (and embedded textual) notations, which makes it difficult to comprehend the diagrams.

In DYNAMO interaction diagrams it has been a criterion to show the overall and important structures of a message tree in a graphical way, including the the nesting of messages and the relevance period of objects, as discussed above. Additional details are dealt with in two different ways:

1. Via pop-up graphics, which show up when the user rests a little while with the mouse button pressed down on some graphical element.

2. Via activation of the message browser or the object browser upon double clicking on relevant graphical elements (see figure 1).

When pop-up graphics is applied on a message, a box appears with a summary of all the underlying message information (see section 2 where these informations are described). The pop-up information disappears as soon as the user releases the mouse button. When applied on an object the similar detailed information appears together with a graphical indication of a possible association or part-of relation involving the selected object. Figure 7 shows an example of our use of pop-up graphics.

The activation of the browsers from the interaction diagram makes it possible to add detailed information conveniently, without cluttering the diagrammatic notation with lots of textual information and other details.

4.3 Interaction techniques.

As presented above, an interaction diagram serves as a presentation of a scenario in a dynamic model. We will now discuss how we have turned the presentation into a tool that also allows us to create and modify a scenario.

Given the intuitively appealing notation of an interaction diagram, it seems important to extend this also to cover the way we edit an interaction diagram. Consequently, we have made a dragging-based direct manipulation interface to the following actions on an interaction diagram:

1. Creation of a new message.

2. Changing the receiver of a message.

3. Changing the place, from which a message is sent.

When a designer wants to add a new message to a scenario, he or she drags a new arrow from the sending object to the receiving object. The name and the parameters are typed directly into the diagram (no indirection in terms of data entering windows is present). Similarly, the designer can change the receiver by dragging the message arrow to another object. It does not make sense to change the message sender in a similar way (by dragging the initial point of an arrow). The reason is that the sender of a message is given implicitly as the receiving object of the surrounding message. If no such surrounding message exists, the sender is a distinguished object called 'root'. Therefore, the natural meaning of dragging the initial point of an arrow is to move the underlying message from one sending context to another.

The dragging-based actions of item two and three above support *redistribution of responsibilities* among the objects in a dynamic model. It is a typical refinement step in a design to decide that a responsibility should be changed from one object to another. Consequently it is important to visualize a holistic view of the responsibilities among the objects. Interaction diagrams (and object graphs) are good in that respect. Furthermore it is important to provide flexible and natural means of interactions for changing the responsibilities. The dragging-based interfaces seems to be very well suited for that.

As it appears from the discussions above, the meaning of dragging depends on the starting point of the action. In order for the designer to feel confidence to "what is going on", the feedback during the dragging signals to the user whether he or she is in the process of making a new message, changing a message receiver, or changing the sending place of a message.

We support the notion of a *current message* and a *current object*. The current message and object are highlighted by use of a distinguished color (red). Editing commands such as cut, copy, and paste operate on the current message and/or the current object. These commands are activated via a traditional pop-up menu, placed on right mouse button. We support different popup menus on messages and objects, respectively.

It is important for the designer to realize how the individual objects are related to the individual messages of a scenario. The roles as sending and receiving objects of a message are easy to spot in an interaction diagram, and the parameter objects are shown in the label of the message. What is missing is the possibility to illustrate the objects which are "functional results" of a message, and the objects which are otherwise provided by the message. The necessary information can, of course, be found in the underlying browers, but it is often too time consuming and awkward to dig the information out from there. The information is also present in the pop-up information boxes, mentioned in section 4.2. By selecting an object we get information about the message, and the place in the message, where the object is provided. Similarly, by selecting a message we list all the relevant object provisions in the object together with other informations about the message (see figure 7).

In order to support an even more direct way of illustrating the objects that are related to a message, we highlight the resulting object (if any) and the objects provided by the message, when the message is selected. The highlighting is done by letting the object names blink a few times when the message is selected. The roles of the objects are clear due to colors superimposed on the object relevance lines. In that way it is very easy to grasp the overall relations between a message and the objects involved in it.

Finally, we will address the way we create new objects via use of the interaction diagram. Using the popup menu in the upper strip of an interaction diagram (where the names of the relevant objects are shown) it is possible to create global objects, local objects, result objects, part-of objects, and associated-from objects. When the new object is related to an existing message (such as the resulting object of message), we use the current message (highlighted in red) as the relevant message. Similarly, when a new object is related to an existing object (for instance being part of it) we use the current object (also highlighter in red) as the relevant object. In that way it is easy to create new objects via the interaction diagram, and it is possible in a very flexible way to specify the "context" of the object.

5 Similar work

In this section we will primarily compare and contrast DYNAMO with SCED [11, 10]. Like DYNAMO, SCED is an environment for dynamic modelling of object oriented systems. SCED supports scenarios as well as state machines. One of the major contributions of the SCED work is to automatically synthesize state machines from scenarios [9]. A comparison between SCED's state machine synthesis and DYNAMO's class and method synthesis can be found in [15]. Here we will compare SCED and DYNAMO with respect to the scenario concept and the tools.

In several respects, scenarios in SCED are more general than scenarios in DYNAMO. SCED allows modelling of multi-sequential interaction among objects. Recall that DYNAMO currently is limited to modelling the behavior of passive objects and uni-sequential interaction. SCED uses the concept of an "event" instead of a "message". Seen from the perspective of an object `O`, SCED describes how `O` *responds to an event* from object `P`, by *causing events* on other objects. Thus, SCED supports a rather abstract object interaction model. This is a contrast to DYNAMO, whose interaction model is much closer to the model applied in a sequential object-oriented programming language.

SCED uses the concepts of *simple scenarios* an *algorithmic scenarios*. Algoritmic scenarios support selection (conditionals) and iteration (repetition). Simple scenarios rely on "plain object interaction" augmented with assertions and a state concept. In [11] it is described how algorithmic scenarios can be transformed to simple scenarios. The scenario concept of DYNAMO is relatively close to the simple scenario concept of SCED. It has been one of the main premises of the DYNAMO work to keep the scenario concept simple and clean. Algorithmic details have not been an issue in DYNAMO dynamic models. (However, we have been concerned with automatic derivation of "algorithm-like" description of methods from a set of scenarios, see [15]). Rather, we want to focus on pure object interaction.

In SCED there is no awareness of object occurrence and object life times. Using SCED the designer must assume that all relevant objects are present during the entire scenario. In DYNAMO there are means to describe how objects appear (become relevant), and how newly provided objects are related to other objects on the scene.

In the starting point SCED was closely related to the scenario and state machine concepts of OMT [18]. Following the tradition in many early OOD methods, the concepts are discussed through the graphical presentations in scenario diagrams and state transition diagrams. (This has changed somehow in UML and OPEN with the appearance of meta models [4]). In DYNAMO it has been an important principle to distinguish the modelling language and the presentation of models, as supported by the tools in the DYNAMO modelling environment. A separate paper has been written about the abstract modelling language [16]. The purpose of the present paper has been to discuss concrete presentations of dynamic models through the various tools in the DYNAMO environment. Here we have followed the tradition of the structure-oriented programming language community [19].

The diagrammatic notation of scenarios in SCED is relatively rich because it includes specific graphical renderings of informal comments, states, actions, repetitions, conditionals, and subscenarios besides the fundamental interaction diagram notation of objects and messages/events. As discussed in section 4.2, it seems problematic to reflect a lot of details through the diagrammatic notation. As already discussed earlier in this paper we have used other approaches in DYNAMO to deal with the details (pop-up graphics and the underlying browsers).

In programming languages the principle of abstraction [21] (section 7.4) is an important means in battling an overwhelming amount of details. The subscenario concept of SCED is an important counterpart in scenario diagrams. By defining part of scenario to be a named subscenario, the space limitation of interaction diagrams are alleviated in the vertical "message direction" as well as the horizontal "object direction". The reduction of space in the horizontal direction is due to the observation that objects that only are used internally in the subscenario need not be presented in the scenario, which refers to (applies) the subscenario. In DYNAMO we do not yet support a scenario abstraction mechanism.

A The abstract grammar of the DYNAMO modelling language

In this appendix we show an BNF abstract grammar of the DYNAMO modelling language.

```
<dynamic-model> ::= <initial-scene> <scenario>-list

<initial-scene>       ::=  <object-provision>-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>

<keep-as-part> ::= <containing-object>

<keep-as-associated> ::= <associated-from-object>

<result>     ::=  <effect-result> | <object-id-result> | <object-provision-result>

<effect-result> ::= <post-situation> <effect>

<object-id-result> ::= <post-situation> <object-id>

<object-provision-result> ::= <post-situation> <object-provision>

<actual-parameter> ::= <object-id> | <object-provision> | <informal-parameter>
```

As part of the meta-language, we use `<x>-list` as a notation of zero, one or more instances of `<x>` in a list structure. The meaning of syntactic categories (nonterminals) is discussed in section 2 of this paper. The syntactic categories without a left-hand-side are either strings/symbols or just atomic terminal informations. As discussed in section 3, the syntactic categories with a left-hand-side definition corresponds to either CLOS classes that aggregate other syntactic categories (constructors) or CLOS (super)classes with a number of specializations (alternatives). Notice that `<post-situation>` is a good candidate of being elevated to an attribute of the syntactic category `<result>`, because the post-situation is part of all kinds of results.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools.* Addison-Wesley, 1986.

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

[3] R.D. Cameron and M.R. Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems*, 6(1):20–54, 1984.

[4] Ratioinal Software Coorporation. UML sematics version 1.0. Available from http://www.rational.com, January 1997.

[5] Ioannis "Yanni" G. Gollis. Graph drawing and information visuaization. *ACM Computing Surveys*, 28A(4), December 1996.

[6] 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.

[7] Guy L. Steele Jr. *Common Lisp, the language, 2nd Edition.* Digital Press, 1990.

[8] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley Publishing Company, 1989.

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

[10] 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.

[11] 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.

[12] Ole Lehrmann Madsen and Claus Nørgaard. *Object-oriented environments - the MJØLNER approach*, chapter 19, An Object-oriented metaprogramming system. Prentice Hall, 1994.

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

[14] 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.

[15] 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.

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

[17] T. Reps and T. Teitelbaum. The Synthesizer Generator. *Sigplan Notices Notices*, 19(5):42–48, May 1984.

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

[19] Gerd Szwillus and Lisa Neal. *Structure-Based editors and environments.* Academic Press, 1996.

[20] T. Teitelbaum and T. Reps. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[21] R.D. Tennent. *Principles of Programming Languages.* Prentice Hall, 1981.

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