Tools for Exploration of Dynamic Models in Object-oriented Design

Kurt Nørmark
Aalborg University
Denmark*

Abstract

In the scope of object-oriented design, a dynamic model is concerned with the mutual interaction among objects. In addition, a dynamic model may deal with creation and deletion of objects. As such, dynamic models are in contrast to static models, which typically are concerned with classes and class relations. It is a central hypothesis of this work that designing with concrete objects is easier and more direct than designing with classes, especially in complicated design situations. In addition it has been observed that dynamic models are much more difficult to capture on a sheet of paper than a static model, because of a mismatch between the medium (paper) and the nature of the model (something being "alive"). In the paper we argue that programs, formulated in an ordinary object-oriented programming language, cannot be used for design purposes, because programming languages are concerned with too many details, which are irrelevant for design purposes. Instead we propose a new formalism, called Dynamo, for description of dynamic models together with a set of tools that constitute a dynamic medium, via which the models can be explored.

1 Introduction

Modeling is a key activity in a program design process. In our context, a *model* is a generalized description represented in a stylized fashion, which is used in analyzing or explaining an existing or a forthcoming program. The models which are relevant for program design purposes can roughly be divided into static models and dynamic models.

In this paper, the terms 'static models' and 'dynamic models' have a special, restricted meaning. A *static model* is considered as a source program abstraction with special emphasis on the structural relations among program components. A *dynamic model* is considered as an abstraction of the program execution. Consequently, a dynamic model

*Department of Computer Science, Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark. Internet: normark@iesd.auc.dk. — This research was supported in part by the Danish Natural Science Research Council, grant no. 9400911.

deals with the concrete data objects, the interaction among the data objects, and the creation and deletion of these during the program execution. Classes versus instances of classes may serve as an example of the difference between the elements of the two models. The distinction between static and dynamic models, considered as abstractions, will be discussed further in section 2 below.

In this paper we will restrict the discussion of dynamic models to *object-oriented design* situations. In general, a *design* is a plan for construction. An *object-oriented design* is a design in which classes and methods play a key role in the static design models, and in which objects and messages play a key role in the dynamic design models.

It is probably more difficult to make a dynamic OOD model than a static OOD model. The reason is that a dynamic model reflects the actual dynamics in a program execution, at some rather high abstraction level. Thus, the elements of a dynamic model may appear, may modify themselves and other elements, and may finally disappear as a function of time. In comparison, a static OOD model is typically concerned with the structural relations among classes, and there are no temporal aspects to be dealt with.

One of the ideas behind this work is that dynamic OOD models should be formulated before the central static models, which involve classes and their mutual relations. It is our hypothesis that programmers think in terms objects, object relations and object interaction during the creative phases of the design process. Consequently, it is probably a gain if the dynamic model can be created directly as one of the first models, at the time where the designer is shaping the overall computational idea relative to the objects at hand.

In the OOD literature dynamic models are typically expressed as various kinds of diagrams or graphs. However, it is not easy to embed information about the "temporal dimension" of a dynamic model in diagrams without making the diagrams very complicated. The complications both affect the producers and the readers of the diagrams. Furthermore the complicated nature of the diagrams severely limits the size of the models which can be handled in a realistic way.

It may be argued that programming languages should be used for descriptions of the desired dynamics. Indeed, a program formulated in a programming language certainly captures the dynamics which we want to model. However, using a programming language the designer is forced to care about a great number of irrelevant details seen in relation to the design task. It is important that the designer can formulate the dynamic model in way which he or she feels is natural in relation to the creative line of thoughts in the mind of the designer. A programming language is not directed towards such a purpose.

Based on the observations that both diagrammatic formalisms and programming languages are problematic for expressing dynamic OOD models, we will in this paper discuss a new approach to dynamic modeling in an object-oriented design process. The approach is based on a high-level formalism for expressing dynamic models together with a set of tools for building, exploring and analyzing the models. In this paper we will only make a rough sketch of the formalism and instead concentrate the efforts on a discussion of the

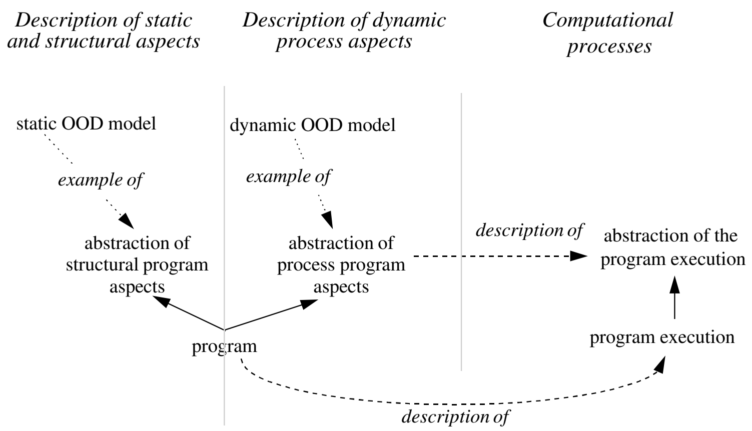

Figure 1: *Abstractions of 'program' and 'program execution'.*

tools.

The collection of tools may, together, be thought of as a *dynamic medium* which matches the properties of the dynamic models. A program document or a diagram on a piece of paper is 'dead' in comparison to a model, which is represented as a data structure in a computer-based tool. The latter may be presented and interpreted in numerous ways, which support the designer in formulating "the right dynamic model" of the solution to a problem, he or she is working on.

Related work can be found in the various books on object-oriented design [14, 3, 11, 7]. Seen in relation to earlier work presented at the Nordic Workshop on Programming Environment Research the work on SCED by Koskimies et al. [8] as well as the work by Salmela et al. [12] are similar to the work described in this paper. The main emphasis of the SCED work is to synthesize state diagrams from object-interactions, as represented by scenarios. Salmela's work is concerned with animation of object-oriented dynamic models.

2 Dynamic OOD models

The main purpose of this section is to summarize the dynamic model, which we have developed. In addition, we will discuss a number of dimensions along which it is possible to characterize dynamic models in general.

At the outset we will clarify our view on dynamic models in contrast to static models. This is done via the concepts and relations in figure 1. The central concepts in the figure are 'program' and 'program execution'. In our view, a program is a description of the program execution. Various abstractions can be defined on the program. Abstractions of the *structural program aspects* are in general known as static models. In relation to this paper, the abstraction of the program execution is of primary interest. The idea with this abstraction is to elevate the actual program execution objects and mechanisms to a level, which is fruitful and adequate for the object-oriented design task. A dynamic model is a description of the abstracted program execution. [1]

It is a central decision of our current work that a dynamic model is a *scenario* more than a program-like description of a computational process. In this context, a scenario is "an outline or a model of an expected or a supposed sequence of events"[2]. Consequently, a dynamic model in our work is a fixed example of object interactions, which can be executed in the mind of the designer. Execution in terms of simulation of the model on a computer is not the goal of this work. With this understanding of dynamic models it is seen that dynamic models are similar to *use cases*, as promoted by Jacobson et al. in [7].

2.1 Concepts of the Dynamo formalism

We will now turn to a summary of the central concepts in our dynamic OOD model, which we call *Dynamo*. The concepts are objects, scenes, and messages.

An *object* is an identity carrying encapsulation of some program state. This is a definition which is similar to the object concept from object-oriented programming languages. In our current version of the design formalism, we do not deal directly with the state of objects. Consequently, we do not describe the data fields of an object. When new objects are introduced it is possible to describe how the new objects are related to the existing objects. The class of the objects are registered. The class of an object does not exist as a description in the dynamic model. The sole purpose of classes is to relate the set of objects, which are instances of the same class.

A *scene* is the set objects which are relevant for the current dynamic model. Objects enter the scene via a mechanism called *object provision*. An object provision is a convenient and somewhat magical mechanism which establish an object exactly at the time it is needed by the designer. An object may actually have existed for a long time, but in the current scenario it first becomes relevant at 'object provision time'. Thus, it is seen that an object provision claims the existence of an object in addition to a certain relationship with the objects, which already have been brought onto the scene. As a simple and practical convention, all objects on the scene have a unique name, through which they can be referred to during a scenario. Other kinds of object references are deemed irrelevant for

[1]From figure 1 it may be noticed that the abstraction of the process program aspects may be reached both from the program and from the program execution. The reason is that a program (considered as the source) contains both structural aspects and process aspects. Consequently, it is possible to focus on the process aspects from an abstraction applied on the (source) program as well as via an abstraction applied on the program execution.

[2]This definition is taken from the *American Heritage Dictionary of the English Language*.

design purposes.

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 are no real sender object, although the "surrounding" may be seen as the sender. In addition to the receiver, it is also possible to pass parameters. Parameters may be existing objects (referred by their names), object provisions (objects of which we claim the existence 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 in the model:

- Provision of a number of objects.
- Passing a number of messages from the receiver of M to other objects.
- Establishing the result of M (in terms of an existing object, an object provision, an informally given value, or an informally described effect on the program state).

We do, in addition, consider the inclusion of explicit object deletion in our dynamic model. This creates symmetry in relation to object provisions. In addition, we think it may be a relevant design decision to state if and when an object becomes irrelevant.[3]

Although it is conceived that a message activates a method from the class of the receiver object, the method concept is weak in our dynamic model. It is known to be the case that the message passings in item 2 from above are located in one single method in the class (say C) of the receiver object R. However, the same message to another object of the class C, or the same message to R with other actual parameters, may cause another sequence of messages from R (for instance because of selections with if-then-else's in the body of the method). These observations are consequences of the scenario-based approach, which we haved followed in our research until now. In conclusion, methods are weak in dynamic models; Methods are parts of the static model.

2.2 Characterization of dynamic models

A dynamic model can be characterized using three independent dimensions:

1. The level of abstraction.
2. The means of expression.
3. The degree of formalization.

[3] It is generally accepted that implicit object deletion via use of some automatic memory management scheme (such as garbage collection) is of great value at the programming level. But this does not necessarily imply that object deletion is entirely irrelevant to the designer.

It is given a priori that the level of abstraction of the dynamic model is significantly higher than that of the actual program execution model. Overall, the level of abstraction should fit the purpose of the task in which the model is used. In our case, the task is object-oriented design. As a special twist, which also affects the desired level of abstraction, we prefer to make dynamic models prior to the static class model.

As mentioned in the introduction, the diagrammatic means of expression dominate in the OOD literature, also with respect to description of dynamic models. This is a contrast to the programming phase of the development process in which the textual means of expression dominates. Our approach in this work is to leave the decisions about 'the means of expressions' open. This is achieved by working on abstract representations of the model, which may be presented either as a diagram or as a piece of text. We come back to this discussion in section 3 of this paper.

On the one hand, a formal description lends itself towards a higher degree of precision than does an informal description. On the other hand, an informal description is often better suited to enhance the intuitive understanding of the model. Both precision and intuition are important for our endeavor. In particular we believe that the designer need to document the intuitive aspects of the dynamic model, in order to promote a human, common sense understanding of the design task at hand. Therefore, we encourage the designer to give intuitive explanations in natural, written language, and we augment the representation of the dynamic model with these explanations. As a consequence we go for a dynamic model in which both formal and informal aspects are present. In some situations this leads to redundancy.

3 Tool support of Dynamo

In our context, a *tool* is an executable program that supports some well-defined task, and which manifests itself to the user via a concrete interface (windows, menus, and the like). As mentioned in the introduction to the paper, we are interested in tools which support the tasks of building, exploring and analyzing dynamic models. All together we think of the tools as a dynamic medium through which dynamic models may be handled in a better way than constructing and reading diagrams on a sheet of paper.

At the outset, it may be relevant to ask if and how the set of tools for dynamic modeling is different from a tool set, which constitutes a programming environment (or a similar environment). After all, we need tools for the rather well-known and common activities such as editing, checking, interpretation, and administration. The answer is that there are both differences and similarities. The main difference is the nature of the model exploration tool, which is unique for our purpose, and central for the approach. As a similarity, the proposed tool set may be seen as a structure-oriented environment in the tradition of systems such as Mentor [5], The Cornell Program Synthesizer [13], Gandalf [6], PSG, [1], and IPSEN [4]. We will now describe the characteristics of the three tools in separation. Afterwards we will address the issue of integration. Finally we will briefly discuss the underlying tool representation.

3.1 Model building

Model building is an editing activity. The purpose of the model builder is to support the designer in creating and modifying a dynamic model. To the extent that a dynamic model can be seen as expressions in a formal language, the well-known range of editing techniques can be applied. In the one extreme, the model can be formulated as text, which conforms to the language, using a text editor. Let us call this the *textual-linguistic approach*. In the other extreme a model may be formulated using an interaction, which is customized especially to this particular task. The result of this interaction is an internal data structure which represents the model. Of that reason we call this the *interaction-datastructure approach*.

Of the following reasons, the textual-linguistic approach is not attractive in our setting:

1. Due to the requirements of both a formal and an informal element of the language (see section 2) it is attractive to be able to emphasize either the one or the other of these elements in a description. At a given time it may be "too much" to face the full degree of redundancy in the model.
2. By using the textual-linguistic approach we are locking ourselves to one particular means of expression (the textual one). As discussed in section 2 we want to be open towards different means of expressions, such as graphs, diagrams, browsers, as well as text. The interaction-datastructure approach supports this well.
3. The interaction-datastructure approach lends itself better towards a tight integration among the model building, exploration and analysis activities than does the textual-linguistic approach. The reason is that the underlying data structures easily can be shared among the tools in the tool set. If we base ourselves on the textual-linguistic approach, some incremental processing of the text is necessary in order to obtain a satisfactory integration with the other activities.

3.2 Model exploration

Model exploration is the central and unique activity of this work. The purpose of 'exploring' is to consolidate the understanding of the dynamic behavior of the object system. It may, to some degree, be possible to gain such an understanding by simply *reading* the model (using some linguistic or diagrammatic presentation). However, we do not think that "simple reading" is enough. We have a number of proposals which we hypothesize complements and enhance the designer's understanding of the object system and its temporal development.

1. The first is *animation* of the interaction among the set of objects. The animation may take several forms. The simple form is a textual tracing, during which the mutual interaction among objects is presented in a structured manner. As a much more elaborate proposal, the interaction may be shown using an evolving graph (along the lines proposed in [12]), where the nodes are object, and where the edges are messages. The evolution of the graph reflects the temporal sequence of object births, object deaths, and message passings.

2. In the second, which may be called *model querying*, the designer retrieves information about the model by asking questions. Typical questions may be "which objects are alive *now*?"; "what is the interaction history of object X?"; And "which classes of objects are represented on the scene *now*?" Some answers may be easily retrieved from the data structure which represents the model. Other answers require a more careful analysis of the model, and as such we see the need for integration with the model analysis tools.

3. The third and final proposal is interaction *browsing* together with *scene browsing*. In this context a browser is a tool that allows the user to explore a hierarchical structure in a disciplined manner [9]. The interaction browser presents the object interaction in a concise and structural form which allows the designer to dive into sub-interactions if he or she wants to. Each interaction is presented with some "context" in terms of the sender object and the sub-interactions. The scene browser presents the set of objects on the scene relative to a given point in the interaction. On each object, shown in the scene browser, it is possible to ask for further information, such as the intuitive explanation of the role of object in the dynamic model.

3.3 Model analysis

Model analysis is concerned with extraction of derived properties from the dynamic model. The derived properties need to be computed based on the information which is directly available in the model. As discussed above, we have seen that the model explorer depends on information which stems from the model analysis. In addition the model analysis tool is used to check for inconsistencies in the model and to extract a partial static model from the dynamic model. Recall, that in our context a static model is concerned with the structural relationships among the classes in an object-oriented design, whereas the dynamic model is concerned with the interaction-oriented relationships among concrete objects. The extraction of static model information from the dynamic model information is based on the dualities between objects and classes, messages and methods, and between aggregation of objects and aggregation of classes.

The key to the generation of a static model from the dynamic model is that all objects are augmented with the class (in terms of a class name), to which they belong. The sole purpose of this is to identify sets of objects with common properties (protocols). If, for instance, object O1 has properties p1, p2 and p3 and object O2 has properties p2 and p4, and if, in addition, O1 and O2 belong to the same class C, then we can conclude that class C has the properties p1, p2, p3, and p4. In that way be can collect properties of classes by collecting properties of objects, which are augmented by the same class. The result is a partial static model. The model is partial because of the following observations:

1. We cannot be sure that the scenarios, on which the dynamic model is based, cover the total number of properties of a class.
2. There are some important, structural relationships among classes which cannot easily be derived from the instances of the classes, namely the specialization-generalization relationships.

The extracted elements of a static model may be pretty printed with respect to a given programming language, and may as such also serve as a rough starting point for the forthcoming implementation process.

3.4 Integration issues

As already mentioned, the integration among the model builder, explorer and analyzer is very important for the success of the environment. During model exploration it is natural to modify og extend the model if and when a need for that is identified; This is model building. Similarly, the model exploration depends on information which is the result of an analysis rather than just direct presentation of information from the underlying model representation.

From a conceptual point of view we think that the entire dynamic medium (or system) is best understood as a number of separate, but tightly integrated tools. From an implementation point to of view the three tools presented above may rather be realized as one single tool.

We represent a dynamic model as a data structure which is quite similar to an abstract syntax tree (an AST). Each node in the tree is an instance of a class in the implementation language, which for the current prototype is CLOS [2]. The generalization-specialization hierarchy among the classes make up the alternation structure in the underlying abstract grammar. The non-abstract classes aggregate grammatical concepts, and as such they make up the constructors of the abstract grammar. Seen in this way, we can talk about a *dynamic modeling language*, which we grammatically have defined at the abstract level.

We have not, until now, defined any concrete, linguistic presentation of models in the dynamic modeling language. As discussed above, we are working with non-textual-linguistic means of expressions, via browsers. However, with the purpose of saving models on files, we have defined an ad hoc “constructor format”, in which a model data structure (an AST) is linearized, and in which each node is represented as a constructor expression. When evaluated, such a constructor expression directly creates the internal data structure.

4 Conclusions and Status

The main contribution of this work is a formalism for definition of dynamic OOD models together with a dynamic medium, through which to work with such models. From a somewhat traditional point of view, the formalism may be thought of a dynamic modeling language. However, we have only formulated the language at a rather abstract level, corresponding to that of abstract syntax trees. The dynamic medium is realized by set of closely integrated tools, the main purpose of which is to support exploration and refinement of dynamic models. More information can be found in [10].

As of May 1996 we are in the process of developing a set of browsers, via which we can build, explore and analyze dynamic models (in terms of a set of related scenarios).

References

[1] R. Bahlke and G. Snelting. The psg system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonay E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification X3J13 document 88-002R. *Sigplan Notices*, 23(Special Issue), September 1988.

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

[4] G. Engels, C. Lewerenz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments part I: Tool specification. *ACM Transaction on Software Engineering and Methodology*, 1(2):135–167, April 1992.

[5] Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The mentor experience. Technical Report 26, INRIA, July 1980. *Also in* Barstow, Shrobe, and Sandewall (editors) "Interactive Programming Environments" McGraw-Hill Book Company.

[6] A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transaction on Software Engineering*, pages 1117–1127, December 1986.

[7] 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.

[8] 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.

[9] Kurt Nørmark. Programming environments—concepts, architectures, and tools. Technical Report R-89-5, Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, March 1989.

[10] 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.

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

[12] 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.

[13] Tim Teitelbaum and Thomas Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981. *Also in* Barstow, Shrobe, and Sandewall (editors) "Interactive Programming Environments" McGraw-Hill Book Company.

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