A hierarchical diagrammatic representation of class structure

Citation for published version (APA):

Document status and date:
Published: 01/01/1994

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 20. Jun. 2019
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr. M. Rem
prof.dr. K. M. van Hee.
A Hierarchical Diagrammatic Representation of Class Structure

O.S. van Roosmalen
Department of Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven
e-mail: wsinonno@win.tue.nl

Abstract

A method is described to graphically represent class-structure and class-relations in an object-oriented application. This graphical representation can be used for system-design at various levels of abstraction as well as documentation of existing programs. A hierarchical way of ordering class-diagrams is an integral part of the method.

KEYWORDS: object-oriented design, object-oriented programming, class diagrams.

1. Introduction

In this paper a technique is presented for the graphical representation of the class-structure of an object-oriented system. The technique can be employed for both system-design as well as the description and documentation of existing programs. In particular it visualizes relevant implementation aspects, and as such it can be used to investigate and document design decisions.

A striking feature of an object-oriented application is the occurrence of software objects that have physical (external) counterparts. This is particularly evident in process control applications (c.f. the design example discussed later). In such systems an internal (software) object captures the state and controls the behavior of its external (physical) counterpart. At the same time it raises the abstraction level of the internal interface of the physical object. The view that is adopted here is that the bare physical object and the corresponding software object together form the dressed physical object. The requirement specification of the system describes the intended behavior of this dressed object. The analysis of the specification may result in an object-model that can directly be used in software design. In fact, such a model may already contain certain design decisions, for instance with regard to object responsibilities. Therefore, analysis and design can often not be clearly separated and it is certainly advantageous to employ the same modeling technique in these two activities.

The present method can be applied at various levels of abstraction and can be used for analysis as well as design. However, it offers particular advantages when applied to program-design since an ultimate refinement of a graphical class-model can be directly translated into an implementation framework. Naturally, such a framework can be exploited most efficiently when an object-oriented implementation language, e.g. Eiffel or C++, is employed. A precise semantics can be attached to the graphical primitives since they are related to language constructs and aspects of program structure. The designer has the freedom to choose the details of this relation and it may involve specific elements of his implementation language. Care should be taken, however, that language dependence is only introduced in a class-model during the final design iterations.
Implementation aspects related to concurrency and distribution depend strongly on the available language-primitives and on the execution platform on which a system is to be run. Therefore, when a conceptual model is constructed, it is more appropriate to postpone attaching a particular concurrency and communication behavior to the objects and their classes. This behavior can be selected later on the basis of the timing and concurrency requirements of the application and the mechanisms that are available. For that reason, the graphical modeling technique presented here does not contain any handles to express communication and concurrency aspects. These handles could be added in accordance with the language and execution environment.

The graphical method proposed here contains many of the aspects that are present in other graphical object-modelling techniques, e.g., those described by Booch [1] and Rumbaugh [2]. This is not surprising, since all these techniques are tuned to the basic principles and goals of the object-oriented paradigm. This is to say that all these graphical representations attempt to, one way or another, describe the structure and behavior of classes and their instances. What distinguishes the different methods is their relative emphasis on the various design and analysis aspects. The weight given to the various system features and the way they are viewed may depend on the application domain, it may change with the design phase and it may depend on the taste of the designer.

This work has been much inspired by Booch's method. His approach, though, focuses on class-relations: message types and object-communication paradigms. Class-structure is underexposed. The method of Rumbaugh offers a better balance between these two aspects. The method presented in this paper places not only equal emphasis on class-structure and class-relations, it also expresses their interplay.

A problem that one experiences when using Booch's and Rumbaugh's class/object diagrams is that they do not clearly express object-dynamics. Although, admittedly, in such diagrams it is hard to explicitly describe dynamic properties of the system, such properties can be implicitly conveyed. One of the major advantages of the object-oriented paradigm is its appeal to the human intuition. Very rich object structure and behavior can be conveyed through the associations provoked by appropriate names for classes and their operations. In this work an attempt is made to exploit this fact by choosing visually distinct icons for class-attributes and class-operations. Names can be attached to these attributes and operations that optimally express purpose and behavior of the objects. In this respect the method presented here differs significantly from the methods of Booch and Rumbaugh. In their diagrams attributes and operations of a class are not graphically shown and one can not indicate the class of an attribute or an object associated with an operation.

Another shortcoming of the design techniques introduced by Booch and Rumbaugh is the absence of guidelines on how class-diagrams should be organized systematically. A strong point of the present method is the possibility to hierarchically structure a design. Fairly strict but straightforward rules are given in this paper for the organization and documentation of design-diagrams.

The paper is organized as follows. In the next section, section 2, the icons for the representation of classes and their relations are presented. In section 3 the hierarchical organization of the diagrams is explained. Section 4 discusses a program-design example, to illustrate the method. Some concluding remarks follow in section 5.

The terminology introduced by Meyer [3] in the context of the language Eiffel will be used in this paper.

2. Representing classes

The basic module in an object oriented application is the class. Since the amount of code in a class is usually kept small, a system encompasses a very large number of classes. This fine grained modularity, together with the particular mechanisms for combining the behavior exhibited by different classes, produces the favorable extendibility, reusability and maintainability properties of object-oriented applications. A major disadvantage of such fine grained structure, though, is a fragmentation that is particularly
disturbing in view of the control-flow aspects: it is often difficult to follow and understand the control flow from the local view offered by one class. The graphical technique proposed in this paper may alleviate part of the problems in this regard because a class is always considered in the context of other classes and various types of relations between the classes are depicted.

An application written in a pure object-oriented language, such as Eiffel, consists of a number of class-declarations. All other programming constructs are contained within these declarations. Although in some languages, like C++, functions and objects may be declared that are not attached to a class this is only a practical point and not of conceptual relevance. Conceptually, such objects and functions can always be appropriately assigned to a class. The ancestor of all classes within the application (such as the Eiffel-class ANY) or the root-class are often good candidates. The diagrams introduced here will deal with the class as basic and only system-entity. A strict separation of the class and object concepts is maintained.

The structure of a class is specified using other classes. The latter ones are either defined by the user or given in a class-library. Thus, for the graphical representation of a system three aspects may be considered relevant: the structure of a class, its relation to other classes, and the interdependence of class-structure and class-relations. The presented technique, indeed, visualizes features of these three aspects. In the following they are discussed in turn.

2.1 Class structure

A class is a collection of features. A feature can either be an attribute or a routine. Attributes correspond to the data-fields of an instance of the class and the routines define the manipulations that can be performed on these data. The icon for representing a class is given in figure 1. A rectangular box represents the class, its name is placed above the rectangle. Attributes of the class are smaller rectangles either placed completely inside the class-box when the attribute cannot be inspected from outside the class, i.e. when the attribute is hidden, or placed on the boundary of the class-box when it is exported. Similarly routines are represented by ellipses that can, again, either be placed completely inside the class or on its boundary depending on whether the routine is hidden or exported. The name of a feature is written (preferably) in the rectangle or ellipse. By convention, class names are in capitals while feature names are entirely composed of lower case letters.

2.2 Class relations

Two types of relations between classes can be distinguished: the inheritance-relation and the so-called use-relation. Whenever an entity is declared in a class, the class is said to use the class-type of the entity. Three distinct ways of using entities exist: an entity can be (1) an attribute, (2) a variable that is local to a routine, or (3) a routine parameter. A hidden attribute or local entity are cases of use-for-implementation since they pertain to implementation details of a class that are not visible from the outside. An exported attribute and a parameter are cases of use-for-interface for obvious reason. When a class is used for interface the using class may either supply or

![Fig. 1 Structure of the class A.](image)

![Fig. 2 Icons for class-relations.](image)
receive the instance. For example, a function-result is received by the user of the function but may not be supplied. Also, an exported attribute may be specified 'read-only'. An arrowhead attached to the use-for-interface icon indicates such situations where the user may only receive an instance.

An additional complication that may arise with these simple relations is the occurrence of a generic class, i.e. a parametrized class. When an instance of a generic class is employed at least two classes participate in the uses- or inheritance-relation: the generic class and the current parameter-instance. For the generic class involved in a relation a dotted line is introduced that shares its origin with the line that points to the instance of the class-parameter. In figure 2 an example is given of the icon that has to be employed when a generic class with one type parameter is used for interface.

### 2.3 Class-diagrams

To combine information on class-relations and class-structure the icons of fig. 1 and 2 are employed as depicted in the example of fig. 3. An arrow representing an inheritance relation is pointing from the heir to the parent. A line representing a use-relation is made to emerge from the location in the class where the corresponding entity declaration is found. Hence, a use-for-interface line emerges from either an exported attribute or a routine. In the latter case this indicates that the routine has parameters or returns a result. A use-for-implementation line emerges from either a hidden attribute or a routine. In the latter case this indicates that the routine has a local entity. Thus, all possible entity-declaration types have a distinct graphical representation. Information on what type of entity-declaration is appropriate, is implicitly obtained from design-choices of higher abstraction: attribute vs. operation, use-for-implementation vs. use-for-interface, and the direction in which object-identity information is passed.

It should be emphasized that the diagrams introduced at present refer to classes and their properties, only. They are therefore called class-diagrams. Dynamic aspects, like the number of instances (objects) of a class that will be created at run-time, are not explicitly depicted. However, it is often relevant to the understanding of the structure and behavior of an application to show some information on dynamic properties. Particularly information concerning object-visibility is often indispensable. Some additions to make this information explicit are shown in figure 4. An attribute of a class that is shared by all instances is shown as a rectangle with round corners within the class-box or on its boundary. A global object is represented by a box with round corners. Such an object is, in principle, accessible to all classes and no use-instances need to be indicated. This graphical representation of shared attributes and global objects is kept independent of the particular mechanisms that is employed to obtain them, because these mechanisms are language-dependent. For example, shared attributes are obtained in C++ by including the specifier static in the declaration. In Eiffel it is obtained as the result of a once-function. In

![Fig. 3 Example class-diagram.](image)

![Fig. 4 Shared attributes and a global object.](image)
C++ an object is made global by declaring it outside all scopes. In Eiffel a global object is essentially an attribute shared by all classes. It is introduced as a shared attribute of the class HERE (or ANY) which is automatically inherited by all classes in the system.

3. Organizing class diagrams

In the following it is important to distinguish two aspects of a class: its interface and its implementation. The interface is formed by all exported features with their interface. The interface of an attribute corresponds to the interface of the attribute's class. The interface of a routine corresponds to the interface of the classes of the various parameters and the returned result (c.f. figure 5). Although the interface of a class has a recursive structure, it should be small and simple in a well designed application.

Both the inheritance and use-relations imply a hierarchy among classes of an application. There is a distinct difference, however, between these hierarchies. When a class is used only its interface will be visible to the using class. Non-exported features and the implementation of exported routines are of no concern to the user. In contrast, when a class inherits, all its ancestors must be taken into consideration during implementation, not only with regard to their interface, but also their implementation details. In addition, the heir may change the implementation of features it inherits, as long as the interface-specifications and invariants of the ancestors are not violated. This is what Meyer [3] calls the "open-closed" principle: a class is open to its heirs but closed to its users.

During the design-process this implies that a black-box view of the used classes suffices to describe the implementation of the user, but that a glass-box view of all ancestors is required. Therefore the diagrams can be made self-contained when the use-relation is employed to hierarchically structure them. Figure 6 and 7 depict the black-box and glass-box view of an example class, respectively.

On the basis of these considerations the following rules are introduced for organizing a design in a set of separate class diagrams.

Rule 1: Each class is assigned a unique diagram. There its glass-box view is given.

Since ancestors are open to heirs, a glass-box view of a class includes such a view on ancestors. Hence, if a class is inherited, its implementation aspects may be visible in more then one diagram.

---

**Fig. 5** Interface of a class C.

**Fig. 6** A class A as a black box.

**Fig. 7** Class A from Fig 6. as a glass box.
Rule 2: If the glass-box view of several classes is given in a single diagram, then they must be related through inheritance.

Rule 3: All used classes are presented as black boxes, i.e. only (the relevant parts of) their interfaces are shown.

The diagrams are ordered according to a numbering scheme that follows the use-hierarchy closely. The highest level in the use-hierarchy, level 0, captures the interface of the root class. The description of this interface is only of relevance when the class is intended for use as part of another application. The implementation of the root is described at level 1. This implementation is achieved through the direct use of other classes. These classes are opened at a lower level in the hierarchy, say level 2, where it is shown how their behavior is produced using yet other, say level 3, classes. (Note that a level number is higher when the level is conceptually lower). At every level one can check that the interface specification of the opened class can be derived from the interface specification of the used classes. Thus, a diagram constitutes a more or less self-contained view of the system. When a top-down design strategy is employed, the interface specification of a class at level \( n \) is followed by the construction of its implementation at level \( n+1 \), while, at the same level a specification is produced of the interfaces of the classes used for the implementation. The following rules describe the numbering scheme.

Rule 4: Each diagram has a unique number consisting of two components: a level number and a sequence number within the level.

Rule 5: Each class carries the number of the diagram that is assigned to it for providing its glass-box view. This number is given with every occurrence of the class.

The diagram in which the implementation of a class is described can thus be readily found. In this diagram the implementation aspects are shown in relation to ancestors and (the relevant part of) the interfaces of used classes. To keep a diagram comprehensible, the ancestors must either be opened in the same diagram or they must have already been opened at a preceding level or at the same level in a preceding diagram. This has some implications for class-numbers:

Rule 6: A heir has a level/sequence number combination that is higher or equal to that of its ancestors.

Rule 7: A used class has a higher level number than its user.

Sometimes a mutual use-relation exists between classes (or even a self-use relation, c.f. fig. 13). In such cases it may be preferred to open these classes in the same diagram (thus departing from rules 2 and 7) or at least at the same level (still giving up rule 7).

A final rule describes the additional information that has to be supplied with each diagram:

Rule 8: Each diagram must be accompanied by a textual description of implementation-aspects, a description of the interfaces used, and ideally a rationale for the design-choices made.

Note that different black-box views of a class may be present in different diagrams (e.g. class HEAT in figures 9 and 11). As long as its name and number do not change, this should not lead to any confusion. The implementation of such a class should, of course, include all interface-requirements. Naturally, all these requirements must be consistent.

4. Example: Home Heating System

To illustrate the described graphical technique, a design-problem as described by Booch in reference [1] has been worked out in detail: the Home Heating System (HHS). In this section a brief description of the problem is given and the most important design aspects and decisions are mentioned. In the appendix the final version of the class diagrams and the implementation in Eiffel is given. This implementation is carried out up to the level of sensors and actuators.

The Home Heating System concerns the control software of a conventional heating system. The system heats a number of rooms in a home. Each room contains a device for measuring temperature and contains radiators.
with valves that can be opened or closed by the software controller. To complicate matters slightly, a room is only heated when it is occupied. A sensor is present to detect such situations. It is even possible to record the living pattern of a room, to anticipate heating requirements and bring the room to the desired temperature just before it is occupied. The furnace produces the heat that is transported to the radiators. It has to be started and stopped by the software according to the heat-requirements in the building. An activation and deactivation protocol is to be followed. Various dials switches and indicators are present to enable human operators to communicate with the system. Some basic error handling must also be performed by the controller.

For a more detailed description of the requirements the reader is referred to Booch [1]. In [1] Booch also discusses the inconsistencies and shortcomings of the informal requirement specification.

For the sake of simplicity, the present design is based on a subset of the requirements as described in [1]. In particular error handling and the maintenance and use of a weekly living pattern are omitted. It is attempted, though, to produce a design that can be straightforwardly extended with these aspects.

The design is approached in two steps. First a conceptual model is built that is kept as independent as possible from an implementation. Structure and relations between classes are based on the structure, the behavior and the conceptually reasonable responsibilities of dressed objects. What is 'reasonable' in this context can often be discovered by thinking about removing or replacing a physical object. The need for a behavior or responsibility should disappear or change with it. In this first conceptual model no assumptions will be made concerning the concurrency behavior of the classes. This is an aspect that will get attention in the second step. There typical implementation aspects will be considered such as concurrency and control-flow, selection of appropriate container classes and efficiency. The design and implementation resulting from this second step is presented in the appendix. The conceptual design is discussed in the remaining part of this section.

The top level class (root-class) in the system is called HHS (fig. 8). It stands for the entire system with all its functionality. The home heating system consists of two parts, the home and the furnace. The main function that the system has to perform is to regulate the flow of heat from the furnace to the house. According to the requirements information on the availability of heat must be passed from furnace to the home: radiator valves in a room may only be opened when heat is available (in most heating systems this is not a necessity, valves can be opened when heat is required). In the design this is achieved by introducing the abstraction HEAT_AGENT that captures producers and consumers of a shared heat-resource, modeled by the object HEAT. The only task of the regulate activity of the HHS class is to observe changes in the requirement of the home for heat and activate or deactivate the furnace accordingly. The implementation of the routines of the HHS can be sketched in pseudo-Eiffel as follows.

Create is
  do
   home.Create;
   furnace.Create
  end;
regulate is
  do
   when home.heat_required
    do furnace.activate end;
   when not home.heat_required
    do furnace.deactivate end
  end

Fig. 8 Diagram 1.1 of the home heating system.
The way the heat-requirement of the home is determined is of no concern at this level and is delegated to the home. Note that the details of the control-flow aspects are swept under the rug through the use of an ad hoc introduced when construct with the meaning that an action has to be undertaken (the do-part) when the abstract condition (the when-part) becomes true.

At the next level, level 2, the implementation of the home and the furnace are considered. Figure 9 describes the class HOME. A home is in this context nothing more then a collection of rooms. Each room has its own heat-requirement. The function of the home is to determine its requirement for heat from that of the individual rooms. Heat is required if and only if any of the rooms require heat (the routine any of the class SET expresses existential quantification). The functionality of the feature of the class HOME can again be expressed in an Eiffel-like manner.

```eiffel
heat_required : BOOL is
do
  when rooms.any.heat_required
    do Result := true end;
  when not rooms.any.heat_required
    do Result := false end
end
```

The class ROOM contains more interesting aspects and is given in figure 10. It contains the sensors to determine the desired and

Fig. 9 Diagram 2.1 of the home heating system.

Fig. 10 Diagram 3.1 of the home heating system.

actual temperature in the room. There are two principal activities that may take place in a room-object: it can compute its requirement for heat and it must manage the heat-flow into the room by opening and closing the radiator valve according to the availability of and the need for heat. The implementation of the routines of this class can be found in the appendix.

If a weekly living pattern is to be maintained for a room a specialization of the class ROOM can be introduced with a different implementation of the routine heat_required then the above given default one.

The class FURNACE is given in figure 11. The routines activate and deactivate set the boolean heat_requested. The furnace manages the actuators according to this setting and to the information obtained from the sensors.

At this point it is appropriate to discuss possible difficulties that may appear when an implementation is obtained as a single Eiffel-system with a single thread of control. When there is only one thread of control, polling must be used to obtain sensor-information, however, no busy waiting must occur for sensor events that may take a long time to occur. Activities in any of the objects found at run-time must be carried out in such a way that no undesirable time-delays will be present. For example, the situation in a room
must be detected and the actuators there must be managed independently of the activities of the furnace: it is undesirable that a room is not managed for a considerable period of time because the furnace is busy waiting for the water to reach a certain high temperature. It can be inferred from the requirement specification that the controller of the furnace must wait regularly for the occurrence of external events. Times involved are of the order of seconds or possibly even minutes.

To resolve potential timing problems, the furnace is modeled and implemented as a state-machine to which control is passed at a high frequency to consider updates for possible state-changes. (It is not necessary to treat any of the other objects as state machines. The rooms, for instance, can be appropriately managed by calling the manage routine regularly). The working of the furnace is most easily described with the help of the state-diagram in figure 12.

Fig. 11 Diagram 2.2 of the home heating system.

The furnace has four states: inactive, blower_on, boiler_on and active. In the inactive state the furnace is off. When it is activated the blower is switched on. Only after the blower has reached its speed, the boiler can be switched on. When the water temperature of the boiler has reached a certain value the furnace is considered to be in the active state. In figure 12 the transitions between the furnace states are labeled by the conditions that need to be satisfied for the transition to take place and the actions that are carried out on the transition: [condition]/actions. The manage-routine of the furnace (see figure 11) could directly implement the behavior given by this state-diagram. However, to promote reusability a state-machine abstraction is introduced as given in figure 13 that will be inherited by the class FURNACE. The update routine of STATE_MACHINE replaces the routine manage. It changes the current state of the furnace to its successor, which is identified by the routine next of this state. (This routine is deferred since its functionality is state dependent and there is no default behavior).

update is
do state := state.next
end
Specializations of the class STATE must be introduced according to the transition diagram in figure 12. Since each state must identify its successor, states must be visible to one another. Therefore, shared attributes in the class FURNACE_STATE are introduced and each state inherits a use relation with the others from this class. The states must also share the furnace’s sensor- and actuator-objects on which they have to perform operations during a transition. These are provided by the class HARDWARE. The final class-diagrams and implementation along these lines are given in the appendix.

5 Conclusion

In this paper class diagrams are introduced that can model the static aspects of an object-oriented application. A design example has been given to illustrate use and organization of these diagrams. The example also shows the close relationship between the graphical icons and program primitives. Nevertheless, the abstraction level of the graphical representation is not lower than that of other approaches [1,2].

During the design of an object-oriented application, it is very important to reflect upon its run-time behavior: how control flows and when, and how many, objects are created. Such aspects are not explicitly modeled by the class diagrams and must be addressed separately. Various techniques exist to model such dynamic properties, e.g. finite-state [4] and data-flow methods [5], but in practice these do not particularly match well with an object-oriented approach. A coherent analysis and design method that integrates various modeling techniques is yet to be developed.

6. References


Acknowledgement

The author would like to thank Dieter Hammer for reading the manuscript and for stimulating discussions.

Appendix

In this appendix the final design and implementation of the Home Heating System are described. Each design diagram is followed by some remarks concerning design decisions and implementation details that have not been mentioned above. Therefore, the appendix must be read together with the general description given in section 4. The Eiffel listing of the classes that are opened in a diagram is given on the pages immediately following the diagram.
Remarks:  
- To simplify matters it is assumed that the structure of the home is stored and can be obtained via the retrieve routine of the class `STORABLE`.  
- A global timer is added. The furnace needs a timer to measure duration of certain states.  
- Routines `update` are added to the interfaces of class `HOME` and `FURNACE`. Control must be passed frequently to the home and furnace via these routines.
-- Home Heating System, Level 1

class GLOBAL
feature
    timer : TIMER is once Result.Create end
end -- GLOBAL

class HHS inherit GLOBAL
feature
    Create is
        do
            home.retrieve_by_name("HOME");
            furnace.Create;
            timer.initialize;
            regulate_heat_flow
        end;
    furnace : FURNACE;
    home : HOME;

    regulate_heat_flow is
        do
            from
                until true
                loop
                    home.update;
                    if home.heat_required then
                        furnace.activate
                    else
                        furnace.deactivate
                    end;
                    furnace.update;
                end
            end;
end -- HHS
Remarks:

- The abstract class `SET` is replaced by the class `LIST` which is actually present in the Eiffel library.
- The attribute of `HEAT_AGENT` is changed to type `BOOL_OBJ`. Just `BOOLEAN` can not be used because shared attributes can not be of simple type.
- The update-routine is also added to the class `ROOM`.

Fig. 15 Diagram 2.1.
-- Home Heating System, Level 2

class HEAT_AGENT
feature
  heat_available : BOOL_OBJ is
    once
      Result.Create
    end;
end -- HEAT_AGENT

class HOME
export update, heat_required, repeat
inherit STORABLE; HEAT_AGENT
feature
  Create(room_list : LIST(ROOM)) is
    do
      rooms := room_list;
    end;

  rooms : LIST(ROOM);

  update is
    do
      from
        rooms.start
      until
        rooms.off
      loop
        rooms.item.update;
        rooms.forth
      end
    end;

  heat_required : BOOLEAN is
    do
      from
        rooms.start
      until
        rooms.off
      loop
        Result := Result or rooms.item.heat_required;
        rooms.forth
      end
    end;
end -- HOME
For each state a class is introduced which inherits `FURNACE_STATE`. In this way all states of the furnace are visible to one another (they inherit a use relation with each other). Each state implements the deferred next routine of the `STATE` abstraction.

The state attribute of the furnace, obtained from `STATE_MACHINE`, contains the current state. The routine `update` asks the current state for its successor.

The routines `activate` and `deactivate` switch the shared `BOOL_OBJ` `heat_requested` on and off. The state objects can detect the value of `heat_requested` and act accordingly.

Remarks:

For each state a class is introduced which inherits `FURNACE_STATE`. In this way all states of the furnace are visible to one another (they inherit a use relation with each other). Each state implements the deferred next routine of the `STATE` abstraction.

The state attribute of the furnace, obtained from `STATE_MACHINE`, contains the current state. The routine `update` asks the current state for its successor.

The routines `activate` and `deactivate` switch the shared `BOOL_OBJ` `heat_requested` on and off. The state objects can detect the value of `heat_requested` and act accordingly.
class STATE_MACHINE export update
feature
  update is
do
    state := state.next
end;

  state := STATE;
end -- STATE_MACHINE

class FURNACE_STATE
inherit HEAT_AGENT
feature
  heat_requested : BOOL_OBJ is once Result.Create end;
  inactive : INACTIVE is once Result.Create end;
  blower_on : BLOWER_ON is once Result.Create end;
  boiler_on : BOILER_ON is once Result.Create end;
  active : ACTIVE is once Result.Create end;
end -- FURNACE_STATE

class FURNACE
export activate, deactivate, repeat STATE_MACHINE
inherit FURNACE_STATE; STATE_MACHINE
feature
  Create is
do
    state := inactive;
  end;

  activate is
do
    heat_requested.on
  end;

  deactivate is
do
    heat_requested.off
  end
end -- FURNACE
The BOOLEAN attribute heat_requirement is added to the class ROOM. The last value of the heat-requirement must be remembered. If the temperature remains within a certain range its previous value will be retained.
-- Home Heating System, Level 3

class ROOM export update, heat_required
inherit HEAT_AGENT

feature

Create is
do
temp_dial.Create;
temp_sensor.Create;
occupancy_sensor.Create;
radiator_valve.Create;
--- heat_requirement := false -- is the default
end;

temp_dial : CONTINUOUS_SENSOR;
temp_sensor : CONTINUOUS_SENSOR;
occupancy_sensor : ON_OFF_SENSOR;
radiator_valve : SWITCH;

heat_requirement : BOOLEAN;

update is do manage end;

heat_required : BOOLEAN is local td, tw, tr : REAL;
do
td := temp_dial.value;
tw := temp_sensor.value;
if occupancy_sensor.is_on then
  tw := td
else
  tw := td-5
end;
if (tr < tw-2) then
  heat_requirement := true
elsif (tr > tw+2) then
  heat_requirement := false
end;
Result := heat_requirement
end;

manage is
do
  -- The radiator valve has to be set
  if heat_requirement and heat_available.set then
    radiator_valve.on
  else
    radiator_valve.off
  end
end

end -- ROOM
The class `HARDWARE` provides the states with shared sensor and actuator objects.

A state must measure the time that elapsed since it was entered. The features `enter` and `entry_time` have been introduced in the class `STATE` to make this possible.

The timer provides a count of the number of seconds that have elapsed since system-initialization.
class HARDWARE

feature

ignitor: ON_SWITCH

is once Result.Create end;

temp_dial: CONTINUOUS_SENSOR

is once Result.Create end;

temp_sensor: CONTINUOUS_SENSOR

is once Result.Create end;

heat_switch: ON_OFF_SENSOR

is once Result.Create end;

oil_valve: ON_OFF_SENSOR

is once Result.Create end;

status_indicator: SWITCH

is once Result.Create end;

blower: BLOWER

is once Result.Create end;

end -- HARDWARE

defered class STATE export next, enter

inherit GLOBAL

feature

next: STATE is deferred end;

enter is
do
  entry_time := timer.count
end;

entry_time: INTEGER;

end -- STATE

class ACTIVE export repeat STATE

inherit STATE; FURNACE_STATE; HARDWARE

feature

next: STATE is
d
  Result := Current;
  if not heat_requested.set or heat_switch.off then
    oil_valve.off;
    status_indicator.off;
    heat_available.off;
    blower_on.enter; Result := blower_on
  end
end;

end -- ACTIVE
class BOILER_ON export repeat STATE
inherit STATE; FURNACE_STATE; HARDWARE
feature
next : STATE is
  do
    Result := Current;
    if heat_requested.set and heat_switch.on
    and (temp_dial.value < temp_sensor.value) then
      status_indicator.on;
      heat_available.on;
      active.enter; Result := active
    elsif not heat_requested.set or heat_switch.off then
      oil_valve.off;
      blower_on.enter; Result := blower_on
    end;
  end;
end -- BOILER_ON

class BLOWER_ON export repeat STATE
inherit STATE; FURNACE_STATE; HARDWARE
feature
next : STATE is
  do
    Result := Current;
    if heat_requested.set and blower.at_speed and heat_switch.on then
      oil_valve.on;
      ignitor.on;
      boiler_on.enter; Result := boiler_on
    elsif (not heat_requested.set or heat_switch.off) and
      (entry_time<timer.count+5) then
      blower.off;
      inactive.enter; Result := inactive
    end
  end;
end -- BLOWER_ON

class INACTIVE export repeat STATE
inherit STATE; FURNACE_STATE; HARDWARE
feature
next : STATE is
  do
    Result := Current;
    if heat_requested.set and (entry_time<timer.count+5*60) and
      heat_switch.on then
      blower.on;
      blower_on.enter; Result := blower_on
    end
  end;
end -- INACTIVE
-- Home Heating System, Level 4

class BOOL_OBJ export off, on, set
feature
    set : BOOLEAN;
    off is do set := true end;
    on is do set := false end
end -- BOOL_OBJ

defered class CONTINUOUS_SENSOR export value
feature
    value : REAL is deferred end
end -- CONTINUOUS_SENSOR

defered class ON_OFF_SENSOR export is_on, is_off
feature
    is_on : BOOLEAN is deferred end;
    is_off : BOOLEAN is deferred end
end -- ON_OFF_SENSOR

defered class SWITCH export on, off
feature
    on is deferred end;
    off is deferred end
end -- SWITCH

defered class BLOWER export at_speed, repeat SWITCH
inherit SWITCH
feature
    at_speed : BOOLEAN is deferred end
end -- BLOWER

defered class ON_SWITCH export on
 inherit SWITCH
feature end -- ON_SWITCH

defered class TIMER export count, initialize
feature
    count : INTEGER is deferred end;
    initialize is deferred end
end -- TIMER
In this series appeared:

91/01 D. Alstein
91/02 R.P. Nederpelt H.C.M. de Swart
91/03 J.P. Katoen L.A.M. Schoenmakers
91/05 D. de Reus
91/06 K.M. van Hee
91/07 E. Poll
91/08 H. Schepers
91/09 W.M.P.v.d.Aalst
91/10 R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude
91/11 R.C. Backhouse P.J. de Bruin G. Malcolm E. Voermans J. van der Woude
91/12 E. van der Sluis
91/13 F. Rietman
91/14 P. Lemmens
91/15 A.T.M. Aerts K.M. van Hee
91/16 A.J.J.M. Marcelis


Implication. A survey of the different logical analyses "if...,then...", p. 26.

Parallel Programs for the Recognition of P-invariant Segments, p. 16.

Performance Analysis of VLSI Programs, p. 31.

An Implementation Model for GOOD, p. 18.

SPECIFICATIEMETHODEN, een overzicht, p. 20.

CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.

Terminology and Paradigms for Fault Tolerance, p. 25.

Interval Timed Petri Nets and their analysis, p. 53.

POLYNOMIAL RELATORS, p. 52.

Relational Catamorphism, p. 31.


A note on Extensionality, p. 21.

The PDB Hypermedia Package. Why and how it was built, p. 63.


An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17 A.T.M. Aerts
P.M.E. de Bra
K.M. van Hee

91/18 Rik van Geldrop

91/19 Erik Poll

91/20 A.E. Eiben
R.V. Schuwer

91/21 J. Coenen
W.-P. de Roever
J. Zwiers

91/22 G. Wolf

91/23 K.M. van Hee
L.J. Somers
M. Voorhoeve

91/24 A.T.M. Aerts
D. de Reus

91/25 P. Zhou
J. Hooman
R. Kuiper

91/26 P. de Bra
G.J. Houben
J. Paredaens

91/27 F. de Boer
C. Palamidessi

91/28 F. de Boer

91/29 H. Ten Eikelder
R. van Geldrop

91/30 J.C.M. Baeten
F.W. Vaandrager

91/31 H. ten Eikelder

91/32 P. Struik

91/33 W. v.d. Aalst

91/34 J. Coenen

Transforming Functional Database Schemes to Relational Representations, p. 21.

Transformational Query Solving, p. 35.

Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.


Assertional Data Reification Proofs: Survey and Perspective, p. 18.


Z and high level Petri nets, p. 16.

Formal semantics for BRM with examples, p. 25.

A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.

The GOOD based hypertext reference model, p. 12.

Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.

A compositional proof system for dynamic process creation, p. 24.

Correctness of Acceptor Schemes for Regular Languages, p. 31.

An Algebra for Process Creation, p. 29.

Some algorithms to decide the equivalence of recursive types, p. 26.


The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.

Specifying fault tolerant programs in deontic logic, p. 15.
<table>
<thead>
<tr>
<th>Year/No</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>92/01</td>
<td>J. Coenen, J. Zwiers, W.-P. de Roever</td>
<td>A note on compositional refinement, p. 27.</td>
</tr>
<tr>
<td>92/02</td>
<td>J. Coenen, J. Hooman</td>
<td>A compositional semantics for fault tolerant real-time systems, p. 18.</td>
</tr>
<tr>
<td>92/03</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Real space process algebra, p. 42.</td>
</tr>
<tr>
<td>92/05</td>
<td>J.P.H.W.v.d.Eijnde</td>
<td>Conservative fixpoint functions on a graph, p. 25.</td>
</tr>
<tr>
<td>92/06</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Discrete time process algebra, p. 45.</td>
</tr>
<tr>
<td>92/07</td>
<td>R.P. Nederpelt</td>
<td>The fine-structure of lambda calculus, p. 110.</td>
</tr>
<tr>
<td>92/10</td>
<td>P.M.P. Rambags</td>
<td>Composition and decomposition in a CPN model, p. 55.</td>
</tr>
<tr>
<td>92/13</td>
<td>F. Kamareddine</td>
<td>Set theory and nominalisation, Part II, p. 22.</td>
</tr>
<tr>
<td>92/14</td>
<td>J.C.M. Baeten</td>
<td>The total order assumption, p. 10.</td>
</tr>
<tr>
<td>92/15</td>
<td>F. Kamareddine</td>
<td>A system at the cross-roads of functional and logic programming, p. 36.</td>
</tr>
<tr>
<td>92/16</td>
<td>R.R. Sc1jee</td>
<td>Integrity checking in deductive databases; an exposition, p. 32.</td>
</tr>
<tr>
<td>92/17</td>
<td>W.M.P. van der Aalst</td>
<td>Interval timed coloured Petri nets and their analysis, p. 20.</td>
</tr>
<tr>
<td>92/18</td>
<td>R.Nederpelt, F. Kamareddine</td>
<td>A unified approach to Type Theory through a refined lambda-calculus, p. 30.</td>
</tr>
<tr>
<td>92/20</td>
<td>F.Kamareddine</td>
<td>Are Types for Natural Language? P. 32.</td>
</tr>
<tr>
<td>Authors</td>
<td>Pages</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>F.Kamareddine</td>
<td>Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.</td>
<td></td>
</tr>
<tr>
<td>R. Nederpelt, F.Kamareddine</td>
<td>A useful lambda notation, p. 17.</td>
<td></td>
</tr>
<tr>
<td>F.Kamareddine, E.Klein</td>
<td>Nominalization, Predication and Type Containment, p. 40.</td>
<td></td>
</tr>
<tr>
<td>M.Codish, D.Dams, Eyal Yardeni</td>
<td>Bottom-up Abstract Interpretation of Logic Programs, p. 33.</td>
<td></td>
</tr>
<tr>
<td>E.Poll</td>
<td>A Programming Logic for Fo, p. 15.</td>
<td></td>
</tr>
<tr>
<td>B. Watson, G. Zwaan</td>
<td>A taxonomy of keyword pattern matching algorithms, p. 50.</td>
<td></td>
</tr>
<tr>
<td>R. van Geldrop</td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.</td>
<td></td>
</tr>
<tr>
<td>T. Verhoeff</td>
<td>A continuous version of the Prisoner's Dilemma, p. 17</td>
<td></td>
</tr>
<tr>
<td>T. Verhoeff</td>
<td>Quicksort for linked lists, p. 8.</td>
<td></td>
</tr>
<tr>
<td>E.H.L. Aarts, J.H.M. Korst, P.J. Zwietering</td>
<td>Deterministic and randomized local search, p. 78.</td>
<td></td>
</tr>
<tr>
<td>J.C.M. Baeten, C. Verhoef</td>
<td>A congruence theorem for structured operational semantics with predicates, p. 18.</td>
<td></td>
</tr>
<tr>
<td>J.P. Veltkamp</td>
<td>On the unavoidability of metastable behaviour, p. 29</td>
<td></td>
</tr>
<tr>
<td>P.D. Moerland</td>
<td>Exercises in Multiprogramming, p. 97</td>
<td></td>
</tr>
<tr>
<td>J. Verhoosel</td>
<td>A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.</td>
<td></td>
</tr>
<tr>
<td>K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.</td>
<td></td>
</tr>
</tbody>
</table>


A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27

Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.

A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.

The Design of an Online Help Facility for ExSpect, p. 21.


A Typechecker for Bijective Pure Type Systems, p. 28.

Relational Algebra and Equational Proofs, p. 23.

Pure Type Systems with Definitions, p. 38.


Multi-dimensional Petri nets, p. 25.

Finding all minimal separators of a graph, p. 11.

A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.

GOLD, a Graph Oriented Language for Databases, p. 42.

On Vertex Ranking for Permutation and Other Graphs, p. 11.

Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.

<table>
<thead>
<tr>
<th>Paper ID</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>93/33</td>
<td>L. Loyens and J. Moonen</td>
<td>ILIAS, a sequential language for parallel matrix computations, p. 20.</td>
</tr>
<tr>
<td>93/34</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Real Time Process Algebra with Infinitesimals, p. 39.</td>
</tr>
<tr>
<td>93/36</td>
<td>J.C.M. Baeten and J.A. Bergstra</td>
<td>Non Interleaving Process Algebra, p. 17.</td>
</tr>
<tr>
<td>93/38</td>
<td>C. Verhoef</td>
<td>A general conservative extension theorem in process algebra, p. 17.</td>
</tr>
<tr>
<td>93/41</td>
<td>A. Bijlsma</td>
<td>Temporal operators viewed as predicate transformers, p. 11.</td>
</tr>
<tr>
<td>93/42</td>
<td>P.M.P. Rambags</td>
<td>Automatic Verification of Regular Protocols in P/T Nets, p. 23.</td>
</tr>
<tr>
<td>93/43</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata construction algorithms, p. 87.</td>
</tr>
<tr>
<td>93/44</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata minimization algorithms, p. 23.</td>
</tr>
<tr>
<td>93/48</td>
<td>R. Gerth</td>
<td>Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.</td>
</tr>
</tbody>
</table>
The object-oriented paradigm, p. 28.

Canonical typing and Π-conversion, p. 51.


Graph Isomorphism Models for Non-Interleaving Process Algebra, p. 18.


Time and the Order of Abstract Events in Distributed Computations, p. 29.