HÖGSKOLAN UNIVERSITY

KARLSKRONA
RONNEBY

Research Report

4/95

Area: **Compu ter Science**

Abstract Object State in Real-Time Control

by

Jan Bosch

ISSN 1103-1581
ISRN HKR-RES --95/4--SE

ABSTRACT OBJECT STATE IN REAL-TIME CONTROL

Jan Bosch

Department of Computer Science and Business Administration, University of Karlskrona/Ronneby, S-372 25 Ronneby, Sweden. E-mail: Jan.Bosch@ide.hk-r.se. WWW: http://www.hk-r.se/staff/bosch

Abstract: Traditionally, finite state machines based approaches are used in real-time object-oriented systems development for modelling the dynamic behaviour of an object. Finite state machines based approaches, however, suffer from several problems when used for modelling large and complex objects. As an alternative, an extended object model, LAYOM, and an associated modelling approach is proposed aiming to solve the aforementioned problems and other problems related to *object state.*

Keywords: Object modelling techniques, Object-oriented programming, States, Statecharts, Real-time languages

1. INTRODUCTION

The term "state" in object-oriented systems is used at two levels, i.e. the implementation level and the analysis and design level. At the implementation level, *state* refers to the values of the data parts of an object. During analysis and design, the term is used to indicate the *boolean states* in a state transition diagram.

An object in a real-time system generally has a relevant state and it might respond different to requests of clients, depending on its state. The conventional object model, however, does not provide support for presenting the object state, required by the clients of the object, at the interface of the object. The result is that many software engineers define *get/set* methods to access the instance variables, which, at least conceptually, breaks encapsulation.

General object-oriented analysis and design methods, e.g. (Booch 1994, Rumbaugh *et* al. 1991, Embley *et al.* 1992, Jacobson *et al.* 1992), *use a* finite state machine based approach for modelling the dynamic behaviour of an object. The dynamic behaviour of an object in *a* real-time system, however, consists of several aspects, i.e. the state of the object, its clients, the requested method, the internal concurrency and the time constraints. All these aspects need to be incorporated in the specification of the dynamic object behaviour.

When specifying the dynamic behaviour of a complex, real-time object, however, the traditional finite state machine approach might lead to problems due to the fact that a state transition diagram has to be implemented on *top of* the object model, rather than being uniformly integrated. A second problem is the limited expressiveness of state transition diagrams for dynamic behaviour incorporating all aspects of state, client, method, concurrency and time. The resulting diagram tends to be very complex and large when incorporating all these aspects.

The general observation is that the dynamic object behaviour modelling techniques used at the design level do not match the object model used to implement the design. For instance, although the state transition diagram supports the specification of real-time constraints on states, it is unclear how to translate this into the object model.

As a potential solution to these problems, an extended object-oriented model, LAYOM, is proposed in combination with a modelling technique that uses the clients of the object as the basis for dynamic object behaviour modelling, rather than the object states. LAYOM extends the conventional object model with *abstract object state, conditions* representing client categories and *layers* around the object and object components. A layer represents a structural, behavioural or application-domain specific relation or a constraint on the object behaviour.

The remainder of this paper is organised as follows. In the next section, the problems of conventional object-oriented approaches are discussed. Then, in section 3, the dynamic behaviour of an object is analysed to determine the required expressiveness of an modelling technique and an associated object model. In section 4, the concepts of *abstract* and *active object state are* discussed. The layered object model is described in section 5 and, subsequently, in section 6, an example is presented to illustrate the proposed approach. Section 7 contains an evaluation of the layered object model approach with respect to the discussed problems and some conclusions.

2. PROBLEMS

The term *'state'* in the context of object-oriented system development has many different meanings associated with it. Some designers use it at the implementation level to refer to combined values of the instance variables of the object. Others use it at the design level to refer to a boolean state in a state transition diagram. Both at the implementation level as well as at the design level, problems have been found with the conventional use of object state, but the underlying problem is that the two notions of 'state' are not uniformly integrated and explicitly related with each other.

When modelling object-oriented applications using conventional object-oriented methodologies (Booch 1994, Rumbaugh *et al.* 1991, Embley *et al.* 1992), problems related to the use of *object state* were identified. These problems are related to the use of state during the implementation phase and to the use of state during analysis and design. Below, the identified problems are described.

2.1 Ad-hoc object state access

Clients of an object generally require information about the state of the object. In numerous applications, *get* and *set* methods are used that access the state of the object directly. Some refer to this practice as bad modelling or as design flaws, but others, including the author, believe that it is natural for objects, being representations of real-world entities, to require information about the state of other objects. However, the approach that many designers take, i.e. ad-hoc defined *get* methods accessing instance variables directly, is bad practice. Although accessing instance variables through get/set methods is not considered as breaking the encapsulation of the object according to some authors (Micallef 1988), the encapsulation of the object is *conceptually* broken and the internal implementation is accessed directly through the get/set methods associated with the instance variable. The relation between the interface and the actual implementation of the object is much too strong. An explicit and well-defined approach to defining and designing the state accessible by clients of the object would reduce the relation between the interface and implementation drastically, thereby improving maintainability.

2.2 *Static object interface*

The conventional object model defines a static object interface, i.e. all methods are constantly accessible during the lifetime of the object. However, often a method should not be executed when the object is in a certain state or by a certain client of the object. Currently, this type of behaviour, when identified, is implemented by defining a test at the beginning of the method code. However, this approach has, at least, two disadvantages. First, it can lead to so called *inheritance anomalies as* described in, e.g. (Bergmans 1994). For example, if a software engineer defines a subclass of a superclass that contains a test at the beginning of the code of a method, and decides to change either the method code or the test, the whole method is required to be redefined. A second disadvantage is that the functionality of the method is mixed with other behaviour specifications. Among others, this is undesirable from the perspective of readability and understandability. If the code of a method contains, next to its actual functionality, specifications for state- and client-based access, concurrency constraints and real-time constraints, the code will be very complex due to the mix of the several aspects. Separating the functionality of a method from the specifications for state-based access and other constraints would provide a solution to these problems.

Also during the modelling of an object, the static interface presents a problem. In the real-world, entities present different interfaces to the entities they interact with based on the internal state or the type of client. This has been identified by several authors, e.g. (Aksit and Bergmans 1992, Wirfs-Brock *et al.* 1990, Booch 1994, Rumbaugh *et al.* 1991, Jacobson *et al.* 1992), and is generally referred to as *multiple views* or *roles.* However, when the interface of the object is dynamically changed based on its internal state, the clients need access to the part of the object state that reflects the interface. However, providing direct access to the object state would break encapsulation. This results in a dilemma for which a solution is presented.

2.3 *Unintegrated finite state machines*

The finite state machine (FSMs) concept and its derivatives, e.g. Harel's *statecharts* (Hare1 1987) and object-oriented approaches like *Objectcharts* (Coleman *et al.* 1992) and *ObjChart* (Gangopadhay and Mitra 1993), are by all popular object-oriented analysis and design methods (Booch 1994, Rumbaugh *et* al. 1991, Embley et *al.* 1992, de Champeaux *et al.* 1993), presented as *the* dynamic object behaviour modelling tool. Although FSMs are generally considered to be an intuitive and powerful tool, one can identify, at least, two problems associated with their use in object-orientation.

The definition of a FSM is generally build on *top of* the object-oriented model, rather then being uniformly integrated within the object model. Most methods define states in the state transition diagram that are totally unrelated to the concrete state of the object. The object specification of an object for which an FSM is defined contains, next to its concrete state, i.e. the instance variables, a *boolean* instance variable specification for each state in the FSM. In addition, each state-changing method contains statements to change the FSM states to represent the change in the *concrete* state of the object.

2.4 Expressiveness of finite state machines

Another problem with the finite state machine approaches is that they provide insufficient complexity reducing means when modelling complex and large systems. This is mainly due to the fact that the number of required boolean states while modelling an object might easily grow beyond manageable, just because of the complexity of the object. The object might not "fit" the translation into one or a few orthogonal state transition diagrams, resulting in a complex and difficult to understand model. Also when an object is recursively composed of other objects, the resulting state space of an object could contain many, recursively nested, objects, could, again, become very large, complex and difficult to understand. For example, imagine an object containing 5 objects where each nested object has two orthogonal state transition diagrams, both consisting of 4 boolean states, i.e. relatively simple objects. A simple calculation shows that the top level object, due to the composition, contains 10 state transition diagrams and 40 boolean states. Generally, the top level object generally is not a mere composition, but defines coordination between the composed objects. The designer, therefore, has to deal with all these states and state transition diagrams to define the behaviour of the encapsulating object. Even when composing rather simple objects, the conventional finite state machine approach does not provide sufficient complexity reducing means to be the *basis* of the dynamic object behaviour model in an object-oriented specification.

2.5 Real-time constraints on states

The conventional modelling methods for real-time systems primarily support the designer to specify deadlines on computation in the system. Object-oriented methods, for example, specify deadlines on the period between message reception and the return from the method execution. Examples of these are MPL (Nirke *et al.* 1990), RTC++ (Ishikawa *et al.* 1990), RealTimeTalk (Brorsson *et al.* 1992) and DROL (Takashio and Tokoro 1992). FLEX (Lin *et al.* 1991) extend this with constructs for specifying start times and ordering relations between methods. For example, a function B cannot start before function A is finished. The composition-filters object model (Aksit *et al.* 1994, Bergmans 1994) provides so-called *real-time filters* that change the real-time constraints specified on messages.

However, the major object-oriented methods do not provide support for specifying real-time constraints on *object states.* One can easily imagine states for entities in the real-world that would benefit from explicitly defined minimal and maximum periods. E.g. the heater of a water tank, used as an example in this paper (see section 6), has the requirement that it may not be in the heating state for more than 10 minutes. The lack of support for this type of real-time constraints is a problem of expressiveness for the conventional approaches.

(Embley *et al.* 1992) give a brief example on how to associate a real-time constraint with a state. They annotate states and describe the deadline in the annotation. Unfortunately, no language support or support for converting the analysis model into an implementation is provided.

The finite state machine based approaches, on the other hand, do provide some support for associating real-time constraints with states. (Hare1 1987, Gangopadhay and Mitra 1993, Coleman *et al.* 1992) model this as transitions with the deadline as a condition. When the deadline is reached, the transition fires automatically and the machine goes to the next state. However, non of these approaches provides a clear translation into an object-oriented language model.

2.6 *Active state*

All finite state machine based approaches use transitions to transfer from one static, stable state to the next. Transitions are generally presented as *atomic* state changes within the state machine. However, the current semantics of transitions can not be considered as natural for modelling for two reasons: (1) transitions generally take time to be performed and during this time other events might take place and (2) transitions model state changes, i.e. moving from one stable, i.e. static, state to another, whereas the actual object state before and after the transition might not be static. Some researchers, e.g. (de Champeaux et al. 1993, Rumbaugh et al. 1991), discuss the notion of an active *state*, i.e. a situation in which the object is involved in an ongoing process. Others, e.g. (Embley et al. 1992), use the term 'interruptible activity' to refer to a similar concept. However, no methodology nor any object model supports the modelling of an *active* object state, i.e. a situation where the object state is not static but, for instance, changing at a static rate. Nevertheless, the use of *active state* does increase the expressive power of the state concept.

3. SPECIFYING DYNAMIC OBJECT BEHAVIOUR

In the previous section, several problems related the conventional use of object state and in particular to the use of finite state machines for specifying dynamic object behaviour were discussed. The finite state machine approaches are used to specify the dynamic behaviour of an object. In this section, the dynamic behaviour of an object is analysed in order to understand the modelling problem that finite state machines are proposed for as a solution.

The dynamic behaviour of an object can be defined as the behaviour of an object in response to the dynamics of its environment and its internal structure. These dynamics need to be constrained, if the object is to remain in a consistent state. The environment of an object consists of other objects that can interact with the object by sending messages. The internal structure of a user-defined object consists of nested objects that are possibly active and can send and receive messages. The encapsulating object contains, next to the nested objects, methods that are executed in response to received messages.

One important aspect of the dynamic behaviour of an object is *the reaction of an object to the receipt of a message.* The default behaviour of an object is to search the method with the name that was used as a selector in the message. But in more complex systems, e.g. a real-time control system, where concurrency, real-time constraints and several other matters play a role, the default behaviour is unsatisfactory.

A state transition diagram as defined in (Coleman *et al.* 1992, Gangopadhay and Mitra 1993, Embley *et al.* 1992) defines constraints on the messages that can be accepted in the respective states and on the transitions from one state to another. If the object for which the state transition diagram is defined receives a message not acceptable in its current state, an exception occurs. A transition might have a real-time constraint associated with it that requires the method associated with the transition to be executed within the specified amount of time. If the dispatched method is not finished within the defined period, again an exception occurs. However, as described in section 2, the use of state transition diagrams in object-oriented system construction has problems associated with it. Alternative specification approaches need to be investigated in determine if these problems can be avoided.

The dynamic behaviour consists of several important aspects which influence the object. In this paper, five dimensions of dynamic behaviour specifications are used. Each dimension is discussed as if it was the primary decomposition for the dynamic behaviour specification:

- Method: One can define *for each method,* the dynamic behaviour. For each method, the software engineer defines the region of the object state that needs to be valid in order to execute the method. In addition, the client objects that are allowed to call the method are defined. Finally, the time and concurrency constraints for the method are declared.
- State: A second approach is to define the dynamic behaviour for *each relevant (boolean) state* of the object. For each state, the software engineer defines the methods that can be executed, the client that are allowed to interact with the object in the current state and the time and concurrency constraints that are relevant for the state.
- Client: A less common approach is to define the dynamic behaviour *for each client* of the object. For each client, the software engineer has to define the object state region in which the client can access the object, the accessible methods, the time constraints for requests from the client and the concurrency constraints.
- Concurrency: The fourth possibility is to define the dynamic behaviour primarily from the perspective of *intra-object concurrency and synchronisation constraints.* A typical example of this approach are exclusion sets. An exclusion set defines the methods that can only execute mutually exclusive. The software engineer would, in this approach, associate the object state region that needs to be valid and the real-time constraints with the exclusion set.
- Time: The time-based approach defines the

dynamic behaviour with respect to *the various deadlines in the system and the type of real-time constraint, e.g. hard or soft.* For each deadline, the methods and object state regions for which the deadline is relevant, as well as the concurrency constraints are specified by the software engineer.

Despite that some of the described aspects of dynamic behaviour specification may feel unnatural, each of the five aspects *can* be used as the primary dimension for modelling dynamic object behaviour. Using *time* or *concurrency* would indeed be less convenient in the general case, but *method* and *client* are as appropriate as *state*. Although most readers will experience the *state*-based decomposition as the most natural, because finite state machine approaches are based on *state*-based decomposition, the other alternatives do have their advantages in modelling.

The full expressiveness for defining the dynamic behaviour requires that the software engineer is able to define the dynamic object behaviour for each individual point in the space formed by *Methods* × *States* × *Clients* × *Concurrency* × *Time*.

The term *States* should not be interpreted as the set of boolean states of the state transition diagram associated with the object. *States,* in this context, is defined as all locations in the state space of the object, i.e. the combination of the domains of all instance variables. The fact that this tends to be a very large set of states, does not diminish that full expressiveness would require this because the software engineer can define dynamic behaviour for each location.

4. ABSTRACT OBJECT STATE

In this paper, an alternative approach to dynamic object behaviour specification is proposed. This alternative approach does not suffer from the problems described in section 2. In this section, the first aspect of this approach is described, i.e. *abstract object state.* The abstract object state provides an alternative for the set of boolean states in a state transition diagram. It will be used as a basis for defining the dynamic behaviour of an object, similar to the boolean states that are used as a basis for the state transition diagram.

An object is, traditionally, defined as consisting of an identity, a state and a behaviour. The state of an object 0 is, basically, the *value* of its parts. For example, the state of a Point object containing two integer objects could be $\{$ 1, 5). This object state is referred to as the *concrete* state of the object. The two integer objects encapsulated by 0 create a two dimensional state space, referred to as the *concrete state space* of the object. Each dimension has a domain associated with it. Both dimensions of object 0 are instances of the *Integer* domain, i.e. [-32768,. ,32767]. The state of object 0 can be viewed as a location in this concrete state space. During its lifetime is 0 always in exactly *one* location in its concrete state space.

The *abstract state* of an object is based on the concrete state space of the object and is an abstraction or conceptualisation of the concrete object state. The abstract state can be viewed as the interface between the conceptual state of the object from an external perspective and the concrete state as it is defined inside the object. For the example Point object 0, the software engineer can be define an abstract state *distance* which represents the distance from the home location $\{0, 0\}$, i.e. *distance*= $\sqrt{x^2+y^2}$. The *abstract state* is generally simpler both in the number of dimensions and in the size of the domains associated with the dimensions. E.g. the abstract state of 0 is one dimensional, whereas the concrete state is two dimensional.

The reason for defining an abstract object state is twofold. First, as an object represents a real-world entity, this real-world entity generally has a conceptual, abstract state which ought to be represented by the object. Conventional analysis and design methods model the conceptual state of an object as a collection of boolean states, but this is considered to be too restrictive. Secondly, the conventional approach of using boolean states to model the conceptual state of an object has complexity problems for any but simple objects. Because the boolean states 'ripple' up the composition hierarchy, the number of boolean states easily grows beyond manageable. The finite state machine approaches use the notion of nested and orthogonal state diagrams, but, nevertheless, the number of states can still grow large. What is required is a mechanism that breaks the naive multiplication of states when objects are recursively composed. The notion of *abstract object state* is proposed which defines, at the object interface, an object state only containing the relevant state of an object. The abstract object state, generally, decreases both the number of states and complexity of dealing with object state.

In the remainder of this section, first the basic object model incorporating abstract state is defined. In section 4.2, the basic object model is extended to incorporate *composed objects* in the formal model and a rationale is described. In the last section, the abstract object state is extended with the notion of an *active state* which uniformly integrates with the abstract object state.

4.1 Basic Object Model

An object generally contains nested objects and methods. A nested object is an instance of either user-defined class or of a primitive, system-defined class. However, as all user-defined objects, either directly or through (recursively) nested objects, make use of system-defined classes, this section describes an object that only makes use of primitive classes, i.e. the set of basic data types, e.g. *Integer*, *Boolean* and *Char*. In the next section, the model is extended to incorporate any type of nested object.

These basic, system defined data type classes have a *one-dimensional* state space and a domain associated with the dimension. For example, class *Boolean* has a domain *[false, true]* and class *Integer* has a domain [-32768,. .,32767]. These domains of these classes are elements of D, the set of domains. Do note that the designer is also able to define domains. This is elaborated upon in section 4.3.

A user-defined object composed of n objects, each an instance of a basic data type class, has a larger state space, consisting of n dimensions. More formally, the concrete state of the user-defined object 0 can be expressed as: $S^c_O = d_1 x\ .\ .\ \mathbf{x}\ d_n$, where 0 is composed of n objects, each object o_i with a domain $d_i \in D$, the set of all system and user defined domains. S^c_O is referred to as the *concrete* state space of the object. At any point during its existence, the object is at exactly one point in its concrete state space.

For now, the abstract state space S^a_O of an object 0 is defined as $S^a_O = (s^a_{O_1}, .. \times s^a_{O_m})$, with m $\geq$ 1. A dimension $s^a_{O,k}$, $1 \leq k \leq m$, is defined as $s^a_{O,k}$: $f^{O,a}_k = S^c_O \rightarrow d_i$, where $d_i \in D$. The abstract object state S^a_O is visible on the interface of 0, i.e. it is not encapsulated. However, clients of object 0 can *only* read the abstract object state and they are unable to change it. In the next section the above definition is extended to any type of nested object.

Object 0 has a method set $M_O = (M_{O,1}, \ldots,$ $M_{O,q})$. Each method $M_{O,i}$, $1 \leq i \leq q$ has a, possibly empty set, of arguments $A_{M_{O,i}} = (a_{M_{O,i},1},$ $\ldots, a_{M_{O,i},r})$ with r $\geq$ 0. The *behaviour* $B_{M_{O,i}}$ of $M_{O,i}$ is defined as $B_{M_{O,i}} = S^c_O\ \mathbf{x}\ S_A \rightarrow S^c_O . S_A$ is defined as $S_A = S^a_{a_1} \times \ldots \times S^a_{a_r}$. $S^a_{A_j}, 1 \leq j \leq r$ is the abstract state space of the j-th argument of $M_{O,i}$. A method $M_{O,i}$ can be viewed as a *vector* in the abstract state space of the object, in that it takes the object from a state S^c_O to another state $S^{c'}_O$.

Do note that only the effects of method execution for the object itself are incorporated in the model. The possible effects for the arguments and the context of the object are not addressed.

An example of the every day use of abstract object state can be found in the primitive, system-defined classes. Instances of, for example, an *Integer* class provide their state at their interface, but from the perspective of the instance, this is an abstract state. The concrete state of the instance is expressed in binary bits and the methods of the instance manipulate the binary bits. However, from the perspective of the client of the object, the object shows, at its interface an abstract state and a number of methods manipulating this state.

4.2 Object Composition

The author believes that a composition-based approach to building object-oriented systems is favourable over inheritance-based approaches to system construction. One reason for this is (see (Bosch 1994c)) that when using composition, one can model many types of relations, including inheritance, between objects in a uniform manner. When inheritance is defined as part of the model, it is treated separately from other, equally important, relations. In section 5, LAYOM, an extended object model, is discussed. LAYOM is based on object composition and representation of relations between objects and constraints on the behaviour of the object as object model components. It also provides means to define the abstract state space of an object.

When using a *statechart-based* approach, objects are defined as containing a set of boolean states. When the object is specialised at a later stage, one or more of the boolean states can be redefined as a nested state, i.e. a state that itself contains a state transition diagram. However, as the described approach aims at a composition-based approach, rather than an inheritance-based approach, an object is *composed* with the required functionality from objects that each provide part of the functionality. To make the object more than its parts, so-called *glue code* is used which binds the composed functionality together. However, as the set of states of the composed object basically is the summation of the states of the nested objects, the set might easily grow very large. To aggravate this, generally most of the states obtained from the nested objects are not orthogonal in the new context. This might cause a state explosion, i.e. a larger number of states than one can reasonably deal with.

The concept of *abstract object state* is proposed to deal with these problems. The use of the abstract object state provides a solution to the identified problems for two reasons:

- As the domain of each state (now called *state dimension)* is not limited to boolean, a single state dimension can be as expressive as a (much larger) collection of boolean states. For example, a state dimension with a domain $[1,..,10]$ could replace 10 boolean states.
- When composing objects, each object has defined an abstract state space on its interface. The abstract object state of a nested object provides a *minimal* representation of its relevant state. The state space formed by the abstract states of the nested objects forms the concrete state of the composed object. The composed object defines an abstract state based on its concrete state which again is minimal in representing the relevant state of the composed object. The uncontrolable state explosion associated with the conventional approach is circumvented because the abstract (or visible) state is decreased to its minimum every time an object is composed.

In the following, objects are defined that are not composed of basic data type classes, but might also consist of user-defined objects. An object 0 is composed of a set of nested objects $(O_1^p, \ldots, O_n^p)$, n $\geq$ 1. The concrete state S_O^c of 0 is defined as $S_O^c = S_{O_i^p}^a \times \ldots \times S_{O_n^p}^a$, where each $S_{O_i^p}^a, 1 \leq i \leq n$, is the abstract object state of nested object O_i^p. $S_{O_i^p}^a$ can be a multi-dimensional state space, i.e. $S_{O_i^p}^a = d_1^{O_i^p}$ x $\ldots \times d_m^{O_i^p}$, $m \geq 1$, where $d_j^{O_i^p} \in D$. D is defined as the set of system and user-defined domains. The abstract state space S_O^a of object 0 is defined as $S_O^a = (s_{O,1}^a, \ldots, s_{O,m}^a)$, $m \geq 1$. $s_{O,k}^a$, $1 \leq k \leq m$, is defined as $s_{O,k}^a : f_k^{O,a} = S_O^c \rightarrow d_j$, where $d_j \in D$.

4.3 Active State

In section 2, problems associated with state and state transitions were discussed. In this section the concept of an *active state is* introduced. Although several authors (de Champeaux *et al.* 1993, Rumbaugh *et al.* 1991, Embley *et al.* 1992) have indicated the importance of an active state, to the best of the author's knowledge, no solutions have been proposed. Here, an abstraction is proposed that allows one to model active state in an expressive and powerful manner by basing it on the abstract object state.

The rationale for introducing *active state* is that it provides a solution to two identified problems. One problem is that in the theoretical notion of a finite state machine, state transitions occur in zero time, i.e. no events can occur during a state transition. Most existing approaches take a pragmatic approach to this by allowing the transition to take time, but still assume that the process performing the transition is encapsulated from the rest of the system. This is considered a problem because it does not model the real-world in a *natural*' way. In reality, a transition cannot be seen as atomic, neither in time nor in space.

A second problem is that boolean states that can only be modelled as *static* states, i.e. the concrete state of the object does not change while the object is in a boolean state. However, this is not always sufficient. In certain cases, (a part of) the state of an object is changing at a constant rate. For instance, the temperature related state of a *water tank* object (used in section 6 as an example) could be modelled as consisting of two states, i.e. *hot* and *cold* and two transitions, i.e. *heating* and *cooling.* But these transitions clearly take a long time and modelling the heating and cooling as transitions is considered to be unnatural.

To address this, one can make use of *active states.* An active state is a stable *dynamic* state, i.e. (a part of) the state of the object is changing at a constant rate. One can view this as analogical to derivate functions in mathematics. For instance, a function f(z) = 2 x x has a derivate function f'(z) = 2. A derivate function in mathematics allows one to think of a function as a stable entity. Similar to higher order derivate functions in mathematics active states can represent 'first-order' state changes, but also 'higher-order' state changes.

To illustrate the notion of *active state,* the example in figure 1 is used. In l(a) the heating related state of the water tank is modelled as two states, i.e. *hot* and *cold,* and two transitions, i.e. *heating* and *cooling.* In l(b), two active states are introduced, i.e. *heating* and *cooling,* which represent the state of the watertank in which the temperature of the water is constantly increasing or decreasing, respectively. For certain types of water tanks, this is still not sufficient and in l(c) four more states are introduced. This is done when a transition, e.g. from *cold* to *heating,* cannot be modelled as an atomic transition, either because it takes a relevant amount of time in which other, important events might happen (time) or because the transition might collide with other activities in the object or the context of the object (space).

An active state is defined as part of the abstract object state. Formally, an active state $s_{O,k}^{a'}$ is defined as $s_{O,k}^{a'} : f_k^{O,a'} = S_O^a \mapsto d_i,\ d_i \in D$. The

[1] **Naturalness is difficult to define exact and is often subjective. Here, a model M_a is considered to be more natural than a model M_b when (1) it has a smaller distance (semantic gap) to the concept being represented (less translation) and (2) it models the concept more precise.**

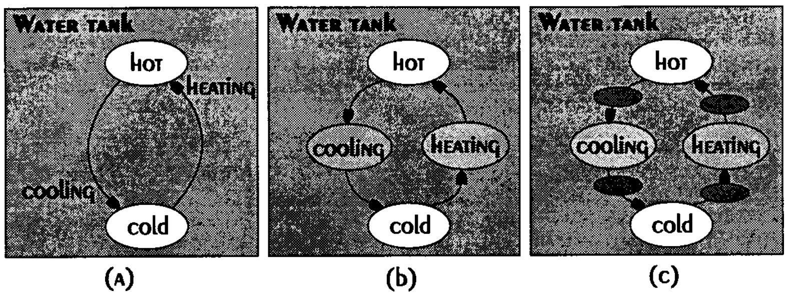

Fig. 1. WaterTank object state

function $\mapsto$ maps the ***derivative*** of the abstract object state S_O^a to a domain ***d.*** Do note that this function is different from the function $\rightarrow$ used in section 4.1. The function $\rightarrow$ maps the concrete state space itself to a domain $\boldsymbol{d} \in \boldsymbol{D}$, whereas $\mapsto$ maps the derivative of the abstract state space to $\boldsymbol{d} \in \boldsymbol{D}$. The derivative of S_O^a is defined as the average of the difference between the last value and the value before that of each selected dimension $d_i \in S_O^a$ divided by the time in between the two updates. If the time since the last update is larger than the time separating the two last updates for a domain d_i, the derivate for that domain is defined as the difference between the last value and the value before that divided by the time interval since the last update.

Formally, a derivative state is a little less trivial. The zero-order abstract state is defined as $S_O^a = (s_{O,1}^a, \cdot, s_{O,m}^a)$, with m $\geq$ 1. A dimension $s_{O,i}^a$, $1 \leq i \leq$ m is defined as $s_{O,i}^a : f_i^{O,a} = S_O^c \rightarrow d_j$, with $d_j \in \boldsymbol{D}$. Due to the message-based, reactive behaviour of an object, the state of object 0 is updated in a discrete manner. This results in a discrete, rather than a continuous, pattern of changes in S_O^a. Rather than defining a continuous derivate function, a, much simpler to implement, discrete derivate function is defined. A dimension $s_{O,i}^a \in S_O^a$ is, in time, defined as a sequence $(s_{O,i}^a(t), s_{O,i}^a(t'), s_{O,i}^a(t''), \ldots)$, where time t refers to the current value of $s_{O\,i}^a$, t' to the time $s_{O\,i}^a$ was updated the last time, etc. For example, $s_{O,i}^a(t')$ refers to the value of $s_{O,i}^a$ in the time interval (t', *t"*). Similar to the base abstract object state the first-order derivate abstract object state is defined as $S_O^{a'} = (s_{O,1}^{a'}, \ldots, s_{O,p}^{a'})$, where $\boldsymbol{p} \geq 1$. A dimension $s_{O,i}^{a'}$, $1 \leq i \leq \boldsymbol{m}$ is defined as $s_{O,i}^{a'} : f_i^{O,a'} = S_O^a \mapsto d_j$, with $d_j \in \boldsymbol{D}$. The function $f_k^{O,a'}$ is generally not the derivate function of the complete abstract object state S_O^a but is defined on a subset $S_O^{a\star} \subseteq S_O^a$ with r elements. $f_k^{O,a'}$ is defined as:

$$f_k^{O,a'}(t) = \frac{\sum \forall s_{O,l}^a \in S_O^{a\star} \frac{s_{O,l}^a(t) - s_{O,l}^a(t')}{min((t'-t''),(t-t'))}}{r}$$

The active state function requires for each domain $d_j \in \boldsymbol{D}$ that is used in derivate function to support the $-, +$ and the $/$ operation. The + and the / operation need to be defined not only within the domain but also when combined with the other domains that are used.

Although the function $\mapsto$ is the default derivation function, the system designer is able to define a different derivation function for each active state. For example, instead of calculating the derivative based on only the last two values of a dimension, the designer might decide to use a more accurate function involving the last three values.

It is possible to define higher-order active states. For example, a second order active state $s_{O,l}^{a''}$ is defined as $s_{O.l}^{a''} : f_l^{O,a''} = S_O^{a'} \mapsto \boldsymbol{d}, \boldsymbol{d} \in \boldsymbol{D}.$ The first-order active state space $S_O^{a'}$ is defined as $S_O^{a'} = (s_{O\,1}^{a'}, \ldots, s_{O\,p}^{a'})$, $\boldsymbol{p} \geq 1$. Obviously, higher order state spaces 'of an object depend on the existence of lower order state spaces. The abstract state of the object S_O^a is defined as $S_O^a = (S_O^a, S_O^{a'}, S_O^{a''}, \ldots)$. The order of the highest order state space is dependent on the relevance of it to the designer. It is not restricted by the model.

The software engineer can define an abstract state, but does not have to do so. Similarly, when an abstract state is defined, it is not required to define a first-order active state space, etc.

5. LAYERED OBJECT MODEL

The layered object model (LAYOM) is an extended object model that is used to investigate ***extended expressiveness.*** The research philosophy is to extend the conventional object model with components that solve certain anomalies, i.e. modelling problems, that the conventional object model suffers from. Examples of these anomalies are described in (Bosch 1994*c*, Bosch 1994*a*, Bosch 1994*b*, Bosch 1995). The layered object model is intended to be an ***extensible*** object model, i.e. it can be extended with new components relatively easy.

An object in LAYOM consists of five major components, i.e. variables (nested objects), *methods, states, conditions* and *layers.* LAYOM does not support *inheritance as a* part of the model, but deals with *inheritance, delegation, part-of* and many other types of relations between objects through *layers.*

The software development approach underlying the layered object model is different from conventional approaches in some aspects. One aspect is that the interface of an object is *not static,* i.e. a service at the object interface might not be available in certain object states and/or to certain clients. Some have objected against this by claiming that it would complicate software development, but the author believes that when modelling a real-world entity, a non static interface is more accurate. In practice, the software engineer often puts tests in the method code to determine the required type of behaviour. These tests often refer to the state of the object or to the state of the arguments.

A logical consequence from having a dynamic interface is that the clients of the object require information about the state of the server object in order to determine when the required interface element can be accessed. However, providing the clients with direct access to the state of the server object would break encapsulation. This is one reason for introducing an *abstract object state* which is an abstraction of the concrete state space of the object and represents the *conceptual* state of the real-world entity that is represented by the object. The *abstract state* is visible at the interface of the object, but it cannot be changed by clients.

A second aspect in which the LAYOM approach is different is that the specification of the dynamic behaviour of the object is not based on the object state as, e.g. the finite state machine. In LAYOM, the dynamic behaviour is defined with the *client objects as* the primary decomposition dimension. Rather than specifying the dynamic behaviour for each relevant boolean state, the dynamic behaviour of the object with respect to each *client category* is defined. As the client categories are not (as much) related to each other as the boolean states in a state transition diagram, the achieved level of decomposition is much higher, thereby decreasing the complexity of the final specification. For each client category, the *state-based access,* the *concurrency* and *time constraints* are specified.

In figure 2, an example of a *water tank* class specification in LAYOM is shown. The *layers* encapsulate the object such that messages sent to the object or sent by the object itself have to pass all layers. Each layer can change, delay, redirect or respond to a message or just let is pass.

```
class WaterTank
  layers
    // structural relation layers
    co-i : Inherit ( Container );
    heater : PartOf ( ElectricHeater );
    temp : PartOf ( TemperatureSensor );
    level : PartOf ( WaterLevelSensor );
  variables
    id : EquipmentNumber;
    height : Integer;
    width : Integer;
    length : Integer;
  methods
    startHeating (stop : Time) returns Boolean
    begin
      ... method code ...
    end;
    stopHeat ing returns Boolean
    begin
      ... method code
    end;
  ...
end WaterTank;
```

Fig. 2. Example class *WaterTank*

5.1 Object model components

A LAYOM object consists, as mentioned, of *variables, methods, states, conditions* and *layers.* In the following sections, each component is discussed.

5.1.1 Variables and methods. The variables of the object are other objects nested within the object in which they are declared. Each nested object has an abstract object state which is visible on its interface. The abstract object states of the nested objects form the concrete state space for the encapsulating object. Variables are declared, as in many other languages, by defining the name and the class of the required object.

The methods of the object are defined as shown in figure 2. The scope of the method is inside-out, and consists of the local variables declared by itself, the arguments of the method, the variables defined by the object the method is part of. Also the objects defined at the same level as object containing the method and all objects at the subsequent levels are visible. Basically, each encapsulation boundary can be 'passed' from the inside, but is 'restricted' from the outside.

5.1.2 Abstract states. A state, as defined in LAYOM, is a dimension of the abstract object state. Each dimension has a domain class and the state expression 'maps' the concrete object state (or a part of it) to the domain associated with the state. Within the LAYOM environment, a domain is required to be a subclass of class Domain. This allows the designer to define his own domains and use them as any predefined domain. However, several predefined domains are available within the system.

As an example, the level sensor of a water tank presents a height in centimeters on its interface. However, the water tank requires the volume, rather the height of the water in the tank. This is defined as:

```
states
  volume returns Liters
    begin
      self.aidth * self.length * level.height
    end;
```

Active states are also supported within LAYOM. An active state is defined based on the static abstract state defined for the object. As described in section 4.3, an active state is defined as a derivate function of (a part of) the static abstract state in time.

An example of an active state is the ***temperature change*** of the water tank. The temperature sensor keeps the current temperature of the water in the tank. The water tank object places this temperature also on its interface. Based on this state it can define an active state *temperature-change* as shown below. Based on the active state ***temperature-change*** a second-order active state *2n-order-temp-change* can be defined.

```
states
  temperature returns Celcius
    begin
      temp.temperature
    end;
  temperatureChange active-on(temperature)
      returns Celcius;
  2ndOrderTempChange active-on
      (temperaturechange) returns Celcius;
```

5.1.3 Conditions. In the previous section, the notion of an abstract object state which is visible at the interface of the object was discussed. However, many researchers have recognised that different clients of an object often should have access to a different part of the interface of the object. For example, in (Aksit and Bergmans 1992) the authors discuss the need for, what they call, multiple views on an object. Others (Wirfs-Brock and Johnson 1990, Wirfs-Brock ***et al.*** 1990, Rumbaugh ***et al.*** 1991, Booch 1994) use the term ***role*** to refer to the object playing different roles to different clients. Also, when using ***use cases*** (Jacobson ***et al.*** 1992), objects involved in multiple use cases need to play different roles depending on the use case which is active. However, most object-oriented languages do not provide support for distinguishing the different client categories of an object.

A real-world example is the following. The example water tank object has three client categories, i.e. ***controller, observer*** and ***equipment.*** The water tank object will behave differently depending on the client category the communicating entity. Object-oriented models, generally, do not allow this type of behaviour.

To facilitate the definition of client categories, the layered object model makes use of ***conditions.*** A condition has an identifier and contains a boolean expression which, taking the message as an argument, defines the discriminating features of a client category.

Formally, a client category $C_{O,i} \in C_O$, the set of client categories of object 0, can be defined as $C_{O,i} : f_{C_{O,i}} = g \rightarrow$ ***boolean,*** where g $\in$ N is a message sent to 0. A message g is defined as g $=$ (0,, O_s, s, ***A),*** where O_r is the object identity (OID) of the receiver of the message, 0, the OID of the sender of the message, s the selector and ***A*** the set of arguments. C_O is the set of client categories of 0 and N is the set of all possible messages. The destination domain of $C_{O,i}$ is the ***boolean*** domain, causing the sender 0, of a message g to either be a member of not a member of a client category $C_{O,i}$.

The ***interface*** I_O of an object 0 is defined as I_O = M_O U S^a_O U C_O. Thus, a message g to an object requests a method execution, i.e. s $\in M_O$, or it request the value of an abstract state dimension, i.e. $s \in S^a_O$, or it requests the evaluation of a message for a specific client category, i.e. s $\in C_O$. In general the evaluation of messages for a client category is done by the ***layers*** encapsulating an object in LAYOM, rather than by client objects.

As mentioned, client categories are defined in LAYOM using ***conditions.*** A condition, just as a state, can be seen as a specialisation of a method. A condition has a single (invisible) argument ***msg,*** which is the message for which it is determined whether its sender belongs to the client category defined by the condition. The return value of the condition is restricted to class ***boolean.***

For example, the ***observer*** client category of the water tank object is defined as follows:

```
conditions
  observer
    begin
      sender.instanceOf(Observer)
    end;
```

5.2 Object Layers

The layers have a very important role in LAYOM. Layers encapsulate an object or an model component, e.g. a state. The first category of layers is referred to as an *object layer,* whereas the second category of layers is referred to as *state layer, method layer,* etc. Each layer is of a type, but is configured towards the specific context in which it is used. A layer type can be used for instantiating object layers as well as object model component layers. In this section, *object layers* are described. In section 5.4, one category of object model component layers is discussed, i.e. *state layers.* Other object component layer categories are not discussed.

An object layer encapsulates an object such that each message sent to or by the object will have to pass the layer. When the message is received by the layer, the message is reified into a passive instance of class Message. The layer has the possibility to handle the message in, basically, any way it likes. It can delay, change, redirect or reject the message or it can decide to execute the request by itself and to return a response.

Layers can be used to represent several aspects of an object specification including the dynamic object behaviour. A layer can be used to model a relation between an object and another object or a constraint on the behaviour of the object.

Layers can also be used to extend the interface of the object. For instance, a layer of type *Condition* can be used to define a new client category which was not defined within the object itself. Similarly, layers of type *State* and of type *Method* can be used, which extend the object with a new (or overriding) state or method, respectively.

From the perspective of a layer, it makes no difference whether it is in between other layers, the most outer or the most inner layer. A object is defined such that at the outside of each layer, one views an object with a interface consisting of methods, states and conditions. An object without layers has an interface consisting of the method names, state names and condition names. When a layer is added to the object, the interface might be extended or restricted, but a client still 'sees' an object with an interface. The layer is transparent, both for client objects as well as for other layers.

A message that is sent to the object has to pass the layers encapsulating the object, starting at the outermost layer and working his way in. Each layer receives the message and inspects it in order to determine what it will do with the message. As described, a layer can delay the message until some precondition is fulfilled or it can change the message contents. Other possibilities are that the layer itself replies to the message or that it rejects the message, generating an exception. Basically, each layer object has the opportunity to handle a message in virtually any way it likes. However, in practice, a layer will just pass most messages on to the next layer and only intercept a subset of the messages. In figure 3, the structure of an object in LAYOM is illustrated.

Analogous to a message sent *to* the object, a message sent *by* the object has to pass all the layers starting at the innermost layer. Again, each layer has the possibility to treat outgoing messages in any way it sees appropriate.

A layer, either in response to a message it needs to evaluate or otherwise, can send messages to the object it encapsulates. In that case, the message has to pass the layers in between the layer sending the message and the kernel object. For instance, a layer can request the value of a state or a condition.

The discussion of the layered object model, in this article, is rather brief. That is because this paper is concerned with states and the state related behaviour of LAYOMin real-time applications. For a detailed discussion of LAYOM see (Bosch 1994*c*, Bosch 1994a).

5.3 Relation Layers

An important category of object layer types is used to represent relations between objects. These relation layers allow a software engineer to convert an analysis and design model, consisting of objects and relations between objects *directly* in the object model. A relation would, in the conventional object model, be implemented on *top of* the model, except for the *inheritance* and part-of relations which are directly supported by the conventional object model. This leads to a number of problems (Bosch 1994c): (1) lack of uniformity, (2) neglected source of reuse and (3) diminished traceability. The use of layers to represent inter-object relations is an approach to solve these problems.

As the term '*relation*' is used in many different contexts and each software engineer has some associations connected with the term, it is important to be explicit about our use of relations. For this purpose, a relation is defined as *any connection from one object (or class) 0, to another object (or class) 0, that influences the behaviour of O_r in some way.* Do note that this definition explicitly demands that the behaviour of the object is influenced by the relation. A second aspect of

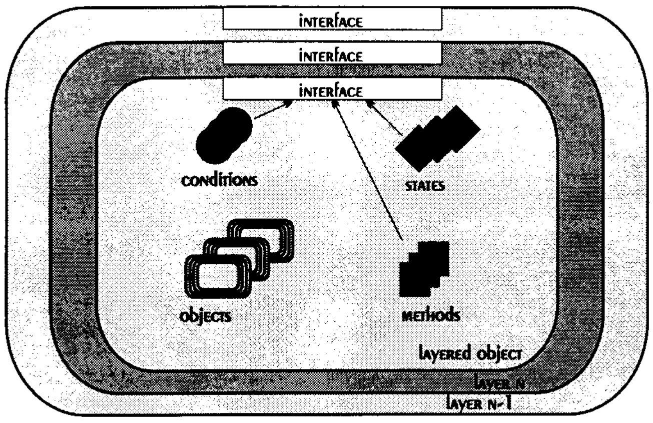

Fig. 3. LAYOM Object

this definition is the unidirectionality of a relation. The rationale for defining a relation as unidirectional is that a relation is associated with the object whose behaviour is influenced by the relation. It is the author's experience that only a small subset of the relation types is bidirectional in terms of this definition. These relations are modelled as two unidirectional relations.

It is the aim of relation layer types to present the concept of relation in a very uniform manner and dealing with each relation type in exactly the same manner. The conversion of an analysis and design model into a set of class definitions is considered to be more natural if relations can directly be expressed in a class definition.

For the discussion of relation types, relations classified into three categories: (1) structural relation types, (2) behavioural relation types and (3) application domain relation types. The relation type categories will be defined in the subsequent sections.

5.3.1 *Structural relations.* Structural relation types creates, as the name implies, the structure of an application. Three types of important structural relations are defined:

- *Inheritance:* This relation type is often seen as the central concept of the object-oriented paradigm. Inheritance allows an class to reuse the interface elements and the internal state defined in the inherited class. The direction of the relation is from the inheriting to the inherited class, as the behaviour of the inheriting class is changed.
- *Delegation:* A second type of structural relation is delegation, i.e. the object can delegate a message to another object while encapsulating this from the sender of the message. Despite the discussion within the object-oriented research community, a number of years ago, concerning the advantages and disadvantages of inheritance and delegation, it is the author's opinion that an object-oriented model should support both relation types. The direction of the delegation relation is from the delegating object to the delegated object, as the behaviour of the delegating object is extended.
- *Part-of:* Perhaps the most fundamental relation in modelling applications is the *part-of* relation. The approach of problem decomposition and solution composition, one of the basic problem solving characteristics of humans, requires the concept of parts. This relation type is directed from the whole to the part, i.e. from an object O_{whole} to an object 0_{part}. The rationale for this is that the behaviour of O_{whole} is extended by its parts and not visa versa.

The above described relation types all provide some form of reuse. The inherited, delegated or part object provides behaviour that is reused by the inheriting, delegating or whole object, respectively. Therefore, next to referring to these relation types as *structural,* one can also define them as *reuse* relations.

Orthogonal to the discussed relation types one can recognise two additional dimensions of describing the extended behaviour of an object, i.e. *conditionality* and *partiality.*

- *Conditionality:* When reusing an object or class, the reusing object might not want to reuse in all states. So, the *abstract object state* is a factor in deciding whether to reuse a particular interface element. When specifying a reuse relation, the software engineer has to determine whether all interface elements of the reused object should be accessible in all object states.
- *Partiality:* The reusing object does not necessarily require all the interface elements of the reused object. Actually, it might even explicitly constrain certain interface elements to be

TABLE 1 Structural relation type identifiers

relation type	inheritance	delegation	part of
default	Inherit	Delegate	PartOf
conditionality	ConditionalInherit	ConditionalDelegate	ConditionalPartOf
partiality	PartialInherit	PartialDelegate	PartialPartOf
conditional and partial	CondPartInherit	CondPartDelegate	CondPartPartOf

reused. So, independent of the object state or the client type, the reusing object only may want to use a subset of the interface of the reused object.

The structural relation types, i.e. layer types, are shown in table 1. For each 'location' in the 3 dimensional space of reuse type, conditionality and partiality, a relation type is defined. Each layer type has its own syntax and semantics. For a more detailed description of the syntax and semantics of structural layer types, see (Bosch 1994c).

5.3.2 *Behavioural relations*. In the previous section, relations were discussed where the object containing the relation is the client, i.e. the object 'extends' itself with the structural relations. In this section the relation types between the object and its clients are discussed. These relation types, generally, constrain the access of clients and the behaviour of the object in some way.

In order to keep the type and number of clients of the object open ended, *client categories* have been defined and for each message is determined whether the sender of the message is a member of a client category. The message is then subject to the behavioural relation(s) defined for that client category.

A relation between the object and a client category can define a number of behavioural constraints. Below, the types of constraints are discussed:

- *Client-based access:* The simplest form of behavioural constraint relation is the restriction of access to interface elements based on the client category. That is, some clients are allowed to access certain interface elements, while this is forbidden for others.
- *State-based access:* In the structural relations, access to interface elements was restricted when the object was in a certain state because the execution of a method or another interface element could possibly lead to an incorrect object state. Here, state-based access is used to restrict the access of certain client categories, generally to favour other clients.
- *Concurrency*[2]: An object, operating in a concurrent environment, can be addressed simultaneously by several clients. To maintain an internally consistent state the object needs to synchronise the concurrent messages.
- *Real-time:* The interval within which a message returns after being received by the object is generally restricted by some upper limit. However, as most systems do not support real-time execution and the deadline is not necessarily hard, the software engineer often relies on the performance of the system, rather than specifying timing constraints. However, although the first-come-first-served ordering might provide optimal throughput, it is generally not the optimal strategy for an application which interacts with a user or other systems, i.e. a real-time system. A mechanism to provide relative ordering of concurrent execution will prove to be very beneficial. Here time intervals are used as part of the behavioural constraint relation.

The layered object model takes a different approach to defining modelling constructs when compared the conventional approaches. Rather than aiming at defining 'clean', orthogonal constructs, constructs are defined such that they correspond to conceptual entities. Hence, supported by the object-oriented analysis and design methods, the concept of a relation between objects is considered a conceptual entity and the different aspects of this relation should, therefore, be defined as part of this relation, rather than as several unrelated orthogonal constructs that combined provide equivalent behaviour.

In table 2, the different relations (or layer) types are shown. Each relation type can be viewed as a location in the space build by the four dimensions defined above. However, a relation always requires a client category to be specified. This results in three dimensions which can be part or not part of the relation. For each combination, a relation type is defined. Again, in (Bosch 1994c), the syntax and semantics of behavioural relation types are discussed in more detail.

[2]Do note that the term 'concurrency' is used in its literal meaning, i.e. concurrent execution of two methods within an object. This differs from the use of the term in e.g. (Bergmans 1994), where the term 'concurrency' includes what in this paper is referred to as 'state-based access'.

TABLE 2 Behavioural relation type identifiers

No *factor*	*One factor*	*Two factors*	*Three factors*
ByClient	ByState	ByStateAndConc	ByStateConcAndTime
	ByConc	ByStateAndTime	
	ByTime	ByConcAndTime	

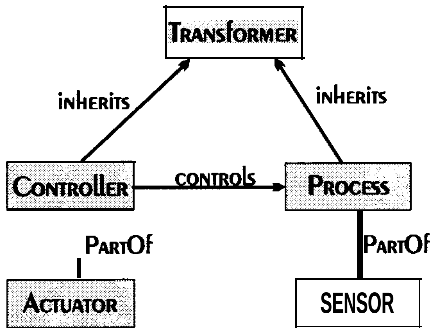

Fig. 4. Analysis and design model

5.3.3 *Application-domain relations.* Next to the general relation types, described in the previous sections, one can also identify application domain specific relation types. Many domains have, next to characteristic classes, for example organised in an application framework, also characteristic relation types that should also be captured in framework and application development.

An example from the domain of *process control* is used. When modelling a process control application framework, the most important objects are the controller and the process. In figure 4, an analysis model of a part of a typical process control application is shown. When modelling relations between objects as first-class entities, the *'controls'* relation between the controller and the process is a typical example of an application domain specific relation type. A *'controls'* relation can be of several types which all behave differently.

In figure 4, class *Controller* has a *controls* relation with class *Process.* Depending on the type of *controls* relation between the controller and the process, i.e. *P, PD, PI* or *PID, a* control relation type is defined between the two classes. For example, in case of proportional control, a *P-control* relation type is selected. Each relation type has its initialisation syntax which has to be defined as part of the class definition containing the relation. Below, the identified relation types are discussed in more detail:

- P-control: The initialisation syntax is: `<identifier>: P-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>)`
  The set method is a method of the *Controller* class that is used to set the required value of the process. The process. input-method is the method at the process object that sets the required value of the process, whereas the process. output-method is the method of the process that gives the actual process value. The multiplication factor is a proportional control constant that indicates how strong a difference between the required and the actual value of the process should be applied in the set value of the process.
- PI-control: The initialisation syntax is: `<identifier>: PI-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>, <integration factor>)`
  All but the last element are the same as the ones defined for *P-control.* The integration factor is a constant for the integrating control. It basically indicates how strongly past deviations from the set value are incorporated in the control signal to the process.
- PD-control: The initialisation syntax is: `<identifier>: PD-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>, <differentiation factor>)`
  Again, all but the last element are the same as the ones defined for *P-control.* The differentiation factor is a constant for the differentiating control. It basically indicates how strongly the angle of the current deviation from the set value is incorporated in the control signal to the process.
- PID-control: The initialisation syntax is: `<identifier> : PID-control(<set method>, <process.input-method>, <process.output-method>, <multiplication factor>, <integration factor>, <differentiation factor>)`
  All but the last two elements are the same as the ones defined for *P-control.* The last two elements are defined as for *PI-control* and *PD-control,* respectively.

These relation types contain all the complex aspects of the respective types of control. If, as in the conventional object model, the control behaviour would be implemented as part of class *Controller,* the control specific behaviour would be mixed with the other functionality of class *Controller.* Also, the functionality of the relation would have become part of the object and the explicit modelling relation between the analysis and design model and the implementation model would have been lost. This indicates the importance of representing a relation as an entity in the object model.

The application domain relation types are, obviously, not predefined. The software engineer can define new relation types and apply these relation types in class definitions as any relation type. When defining a new relation type, it is basically the name, the syntax of the initialisation and the semantics that have to be defined.

5.4 State layers

Next to the *object layers* that encapsulate the whole object, there is a second category of *object component layers* that encapsulates a component of the object model, i.e. a *state, conditions, method* or *nested object*. In this paper, a new object component layer type is introduced, i.e. *StateConstraint*. The reason for introducing this new layer type is related to the problem described in section 2.5, i.e. *real-time constraints on states.*

Although finite state machines approaches provide, on the modelling level, constructs for specifying real-time constraints on states, this is only available for *static, boolean states.* In addition, the conventional approaches do not provide support for implementing this type of behaviour in the underlying object model that is used for implementing the design.

Time constraints on object states are different from the other aspects of the dynamic behaviour of the object in that the other aspects are generally related to the receipt of messages sent by client objects and the subsequent execution of a method. This is not the case for real-time constraints on states. For instance, if an object contains a timing constraint on a state, the time constraint can cause an exception without any messages received by the object. As an example, the water tank object (described in section 6) should not be *heating* for more than 10 minutes. When the deadline is violated, the exception handling should turn off the heater.

Because the real-time constraints on states cannot be specified as part of the dynamic behaviour of an object in relation to its client, the behavioural relation layer types are not suitable for specifying these constraints. This requires a new category of layer types, but the question whether this new category of layer types should be defined for the object as a whole, or that layers should be associated with the object component that is affected by the constraint. The above line of reasoning lead to the definition of the object component layers and, in particular, the *StateConstraint* layer type which is associated with a state within LAYOM.

In figure 5, the notion of state layers is illustrated. Each state can have one or more layer encapsulating it. Currently, only layers of type *StateConstraint* have been defined for states. The state layer is an *active* object that monitors the state and triggers messages according to its specification.

The syntax of the *StateConstraint* layer type is defined as follows:

```
<identifier> :  StateConstraint(<state-region>,
                'Valid' sc-express )

sc-express   :  sc-expr and sc-express
             |  sc-expr
             ;
sc-expr         time-part  action-part

time-part    :  minimally <time>
             |  maximally <time>
             |  every <time>
             ;
action-part  :  onEnter <message>
             |  onExit <message>
             |  on Violation<message>
             ;
```

A layer of type *StateConstraint* defines a *state region,* which is a subdomain of the domain of the state. The layer, being an active object, monitors the state that it encapsulates and notices when the state enters and leaves the state region. The *StateConstraint* layer can trigger messages in response to the state *entering* the specified state region. In addition, it can trigger a message when the state *leaves* the state region. Moreover, it can specify *time constraints* on the *minimal* time period the state is in the state region, on the *maximum* time period the state is in the state region and on the interval in which the state needs to have been within the state region.

A *StateConstraint* layer can be applied to any state of the object, including *active states.* In the next section, examples of the use of *StateConstraint* layer types will be shown.

6. EXAMPLE

To illustrate the issues discussed in this article, the example of a water tank is used. The water tank can be filled and emptied and the water in the tank can be heated by a heating element. It contains two sensors, i.e. a level sensor and a temperature sensor and two valves that are used to fill and empty the tank, respectively. In fig-

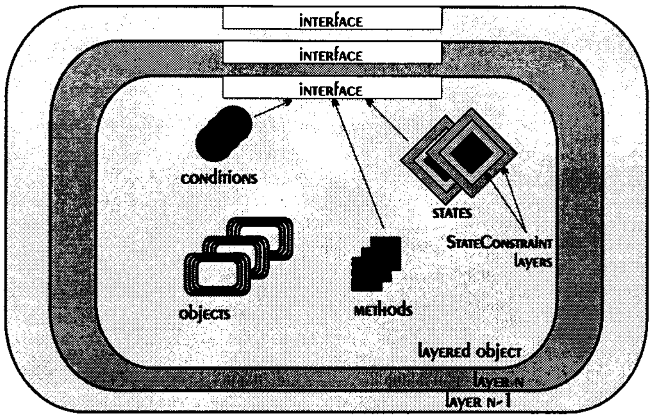

Fig. 5. Layers around states

ure 6, a graphical representation of the example application is given.

When modelling this application using an object-oriented method, the example is decomposed into a number of objects. The water tank object has four parts, i.e. a water temperature sensor, a water level sensor, an input valve and an output valve. The state of the water temperature sensor is represented as a voltage level. The water level sensor contains, as a state, a water height in centimeters. The water tank object has to convert these values in the actual water temperature and water volume, respectively. Similarly, the state of the input and output valve is represented as an open ***percentage.*** The water tank object has to convert these percentages into an input and output flow.

The water tank object has a number of methods on its interface that can be called by client objects. These methods are the following:

- *startFill*
- *stopFill*
- *startEmptying*
- *stopEmptying*
- *startHeating*
- *stopHeating*

In addition, the water tank object has a number of states, that are abstractions of the internal state of the object:

- *waterTemperature*
- *waterLevel*
- ***input*** *ValveState*
- ***output*** *ValveState*

6.1 Dynamic behaviour requirements

The dynamic model of the example application is slightly more complex, due to a number of constraints on the water tank behaviour. These constraints can be categorised into four groups, i.e. state-based, client-based, concurrency-based and time-based constraints. Below, the constraints are described per category.

- **State-based constraints:** These constraints restrict the behaviour of the object with respect to its state. The water tank object has three state-based constraints.
  1. The water tank should never be heated for longer than 10 minutes.
  2. The water tank should not be heated faster than 5 degrees celcius per minute.
  3. The water tank should never be heated while filling or emptying.
- **Client-based constraints:** This constraint type restricts the behaviour of the object with respect to the client object requesting functionality. The water tank object has three categories of clients. The constraints are described according to the class of the client objects.
  1. **Controller:** The controller is allowed to access all the methods and states of the water tank object, except for the emergency method *stopAll.*
  2. **Observer:** Observer objects are only allowed to read the state of the water tank, i.e. temperature, level, input and output flow.
  3. **Emergency:** Emergency objects are only allowed to execute the *stopAll* method.
- **Concurrency-based constraints:** Multiple client objects can access the water tank object simultaneously. These requests need to be synchronised for correct operation. Otherwise, the concurrent execution of two methods might lead to inconsistent object states. Concurrency is specified per client category.
  1. **Controller**: The water tank object requires mutual exclusion of controller threads, i.e. only one active thread.
  2. **Observer:** As observers only read the object state, the observers can execute fully

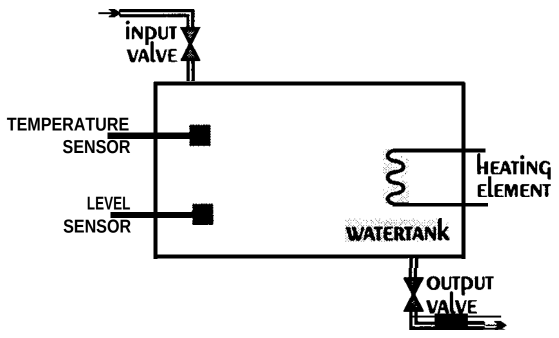

Fig. 6. Example: water tank

concurrent.

3. Emergency: Requests from emergency objects have to be executed mutually exclusive.

- Real-time constraints: Real-time constraints restrict the time frame in which the receiving object can respond to a message. Real-time constraints are primarily concerned with deadlines, but also the start time of the method execution can be specified. The real-time constraints for the water tank are the following.
  1. Controller
     all methods : 1 second ***hard deadline***
  2. Observer
     all states : 10 seconds soft ***deadline***
  3. Emergency
     stopAll: 0.1 second ***hard deadline***

6.2 LAYOM *water tank model*

In figure 2, the structural part of the water tank object, as specified in the previous section, was described, but the dynamic behaviour part is the subject of this section. As mentioned in section 5, the dynamic behaviour modelling in the LAYOM approach is decomposed according to the different client categories an object has. The water tank object has three client categories, i.e. ***Controller, Observer*** and ***Emergency.*** In the layer specifications below, the dynamic behaviour specifications for each client category are specified using the layers of the LAYOM model.

Controller. The layer specification for clients in the ***Controller*** client category is the following:

```
contr: RestrictStateConcAndTime( Controller,
   accept ⋆ except (stopAll, startHeating) and
   1 active-threads and finish-latest 1000 hard,
   accept startHeating when temperatureChange = 0
   and 1 active-threads and finish-latest 1 hard);
```

Observer. The layer specification for clients in the ***Observer*** client category is the following:

```
obsvr : RestrictTime( Observer, accept
   (waterTemperature, temperatureChange, waterLevel,
   input ValveState, output ValveState)
   finish-latest 10 soft);
```

Emergency. The layer specification for clients in the ***Emergency*** client category is the following:

```
emer : RestrictConcAndTime( Emergency,
   accept stopAll 1 active-threads and
   finish-latest 0.1 hard);
```

State Constraints. Two state constraints specified for the water tank object are directly related to the states, rather than to a particular client category. These constraints are specified using layers of type *StateConstraint.*

```
states
  temperature returns Celcius
    begin
      temp.temperature
    end;
  temperatureChange active-on (temperature)
      returns CelciusPerMinute;
    layers
      sc-tcl : StateConstraint( temperatureChange > 5,
      onEnter self.stopHeating);
      sc-tcl : StateConstraint( temperatureChange > 0,
      maximally 600 onViolation self.stopHeating);
    end;
```

7. EVALUATION AND CONCLUSIONS

In section 2, several problems related to the use of object state and finite state machines were discussed. In section 5, the layered object model was introduced as a solution to these problems. In this section, the layered object model is evaluated with respect to the discussed problems.

- Ad-hoc object state access: Because of the explicit definition of an ***abstract object state*** at the interface of the object, the software engineer does not require the ***get*** and ***set*** methods to access the concrete object state directly. The abstract state is a conceptualisation of the internal state of the object and it is accessible, for reading, at the interface of the object.

- **Static object interface:** The object interface of a LAYOM object is typically not static. The layers allow the software engineer to dynamically hide interface elements, depending on the state of the object, the type of client or other aspects of the dynamic behaviour of the object.
- **Unintegrated finite state machines:** This problem can be summarised by stating that the boolean states of a finite state machine are not uniformly integrated within the object-oriented model, but that these states are implemented on *top of* the object-oriented model. The *abstract object state,* which is proposed as an alternative, is fully integrated in the object model. The abstract states are implicitly updated by the object and are used by the layers as *one* of the factors for specifying dynamic object behaviour.
- **Expressiveness of finite state machines:** The concept of *abstract object state* and the client-based specification of dynamic object behaviour results in a richer and more expressive approach to specifying the dynamic object behaviour, when compared to finite state machine based approaches. The specifications for the client categories are more orthogonal to each other than the different states in *a* finite state machine.
- **Real-time constraints on states:** The definition of a new layer type, i.e. *StateConstraint*, allows the specification *and implementation* of real-time constraints on states. Moreover, it supports the specification of real-time constraints on *active states.*
- **Active state:** Active states, i.e. a stable, dynamic object state in which a part of the object is changing, e.g. at a constant rate, have been recognised by several authors as a feature lacking from conventional object-oriented approaches. The notion of *abstract state* provides an excellent basis for defining *active states* of an object. In LAYOM, an object can define both static and active states.

Concluding, the layered object model offers an alternative approach to the development of real-time object-oriented systems which does not suffer from the problems that conventional approaches suffer from. The combination of the *abstract* and *active object state,* the *state layers* and the *client-based* modelling approach circumvent the problems.

LAYOM is supported by a prototype development environment ATOM running on Sun Sparc workstations under the Solaris[3] 2.3 operating system. ATOM uses *threads* and *light-weight processes* for achieving concurrency and Unix[4] sockets for achieving distribution. Support for real-time constraints on messages and states is planned to be added in the near future.

REFERENCES

Aksit, M. and L. Bergmans (1992). 'Obstacles in object-oriented software development'. pp. 341-358. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

Aksit, M., J. Bosch, W. vd Sterren and L. Bergmans (1994). 'Real-time specification inheritance anomalies and real-time filters'. pp. 386-407.

Bergmans, L. (1994). Composing Concurrent Objects. Ph.D thesis. University of Twente. Enschede, The Netherlands.

Booch, G. (1994). *Object Oriented Analysis and Design with applications - 2 ed.*. Benjamin/Cummings Publishing Company, Inc.

Bosch, J. (1994a). Abstracting object state. Research Report 10/94. University of Karlskrona/Ronneby. Ronneby, Sweden. Submitted to Object-Oriented Systems.

Bosch, J. (1994b). Paradigm, language model and method. Research Report 8/94. University of Karlskrona/Ronneby. Ronneby, Sweden. To be presented at the ICSE-17 Workshop on Research Issues in the Intersection of Software Engineering and Programming Languages.

Bosch, J. (1994c). Relations as object model components. Research Report 9/94. University of Karlskrona/Ronneby. Ronneby, Sweden. Submitted.

Bosch, J. (1995). Parser delegation – an object-oriented approach to parsing. In 'Proceedings of TOOLS Europe '92'. Versailles, France. pp. 55-68.

Brorsson, E., C. Eriksson and J. Gustafsson (1992). Realtimetalk: An object-oriented language for hard real-time systems. In 'Proceedings of IFAC International Workshop on Real-Time Programming'.

Coleman, D., F. Hayes and S. Bear (1992). 'Introducing objectcharts or how to use statecharts in object-oriented design'. *IEEE Transactions on Software Engineering* **18**(1), 9-18.

de Champeaux, D., D. Lea and P. Faure (1993). *Object-Oriented System Development.* Addison-Wesley.

[3] Sun and Solaris are trademarks of Sun Microsystems inc.

[4] Unix is a trademark of AT&T Bell Labs

Embley, D., B. Kurtz and S. Woodfield (1992). *Object-Oriented System Analysis - A Model-Driven Approach.* Prentice Hall.

Gangopadhay, D. and S. Mitra (1993). Objchart: Tangible specification of reactive object behavior. In 'Proceedings ECOOP '93'. LNCS 707. Springer-Verlag. Kaiserslautern, Germany. pp. 432–457.

Harel, D. (1987). 'Statecharts: A visual formalism for complex systems'. *Science of Computer Programming 8* pp. 231–274.

Ishikawa, Y., H. Tokuda and C. Clifford (1990). Object-oriented real-time language design. Technical report. Carnegie Mellon University. USA.

Jacobson, I., M. Christerson, P. Jonsson and G. Övergaard (1992). *Object-oriented software engineering. A use case approach.* Addison-Wesley.

Lin, K., S. Liu, K. Kenny and S. Natarajan (1991). Flex: A language for programming flexible real-time systems. In A. van Tilborg and G. Koob (Eds.). 'Foundations of Real-Time Computing: Formal Specifications and Methods'. Kluwer Academic Publishers. pp. 251–290.

Micallef, J. (1988). 'Encapsulation, reusability and extensibility in object-oriented programming languages'. *Journal of Object-Oriented Programming* pp. 12–34.

Nirke, V., S. Tripathi and A. Agrawal (1990). Language support for the maruti real-time system. In 'Proc. 1990 Real-Time Systems Symposium'. IEEE Computer Society Press. pp. 257–266.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991). *Object-oriented modeling and design.* Prentice Hall.

Takashio, K. and M. Tokoro (1992). DROL: An object-oriented language for distributed real-time systems. In 'Proceedings of OOPSLA '93'. ACM Press. Vancouver, Canada. pp. 279–294.

Wirfs-Brock, R. and R. Johnson (1990). 'Surveying current research in object-oriented design'. *Communications of the ACM* **33**(9), 104–124.

Wirfs-Brock, R., B. Wilkerson and L. Wiener (1990). *Designing Object-Oriented Software.* Prentice Hall.