Improving statechart composition and reuse in UML

Sylvain Vauttier & Christelle Urtado

LGI2P - Ecole des Mines d'Alès, Site EERIE, Parc Scientifique G. Besse, 30000 Nîmes

Phone: (33) 04 66 38 70 00 – Fax: (33) 04 66 70 74

Mail: {Sylvain.Vauttier, Christelle.Urtado}@site-eerie.ema.fr

Resume: Composite objects reveal themselves to be very useful in component-based development processes for the modeling, the management and the reuse of component assemblies. Although the composition relationship notion has been widely studied, in order to use the composite objects' structure as a mean to model and manage complex data, few works have investigated the modeling of composite object behavior. Indeed, as far as component based development is concerned, the point is not only the building of complex data structures, but also the combination of the behavior of distinct objects in order to get them work as a coherent entity, which is embodied by a composite object. Classical object-oriented design methods, such as those based on the UML modeling language, do not provide adequate means for efficiently capturing the design of the global behavior of composite objects. As a consequence, they fall short in providing models which possess all the benefits expected from the use of composite objects, such as their ability to abstract complex assemblies. Thus we developed a specific design method for composite objects, called COBALT, which adresses the above issues by allowing a more abstract and declarative design of composite object behaviors. These concepts are provided as an extension to UML, in order to enhance its modeling capabilities for composite objects.

Keywords: component-based development, reuse, composite object behavior, statecharts, UML

1. Introduction

Software engineering is slowly becoming a « real » industry. As in many other technological fields, vendors are beginning to produce software components which, as chips and integrated circuits for the electronics industry, may become distributed on a large scale, through catalogues, and will enable lower-cost, better-quality application development to be achieved from the reuse of ready-to-use, well tested pieces of software. This (r)evolution, sign of a maturing technology, is probably only at its early stage. Nonetheless, it is attested by the growing number of application frameworks and CASE tools which provide support for component based development [2, 15, 20, 25], by integrating component technologies such as the Enterprise Java Beans model [29]. This technological (r)evolution implies a no less significant (r)evolution for the software engineers' work. Indeed, the amount of coding required for the development of an application is reduced, giving an even more predominant role to analysis and design. New tasks appear during these steps: first, the selection of components whose specifications match the requirements of the application, then, the adaptation of these components to the specific context of the application and, finally, the building of a coherent component assembly which implements the target application. This new paradigm entails new specialties in the software engineering field: component provider, who designs components for reuse, application architect, who designs applications by reuse of components, framework or CASE tool provider, who designs technologies and tools providing support for the building and deployment of applications as component assemblies [4, 5, 29, 17].

In this technological context, reuse is the main topic. Software components, which are reusable elements for the development of applications, are designed to be reused as components of larger components or as users of finer grained components — these two faces of the reuse process entails specific constraints on component-based development. Besides, another current trend in the design of software [8, 9, 10], is to consider as a reusable matter not only the code of a component but also the more conceptual elements which have preceded its implementation: the requirements it satisfies and its design. These conceptual elements and their implementation taken together, form a coherent whole which can be used by component-based development tools to provide application architects with different views of components along the different steps of the development process. Such abstract definitions of components enable the tasks mentioned above (component search, selection, adaptation, and assembly) to be carried out. How to reuse software components is definitely a pertinent problem not only during the coding phase but also during the requirement analysis and design phases.

Thus, our work on component-based development has mainly focused on the reuse of the specifications of components in order to make both the building of large and sophisticated component assemblies and the reuse of these assemblies as new components easier [31, 32]. In this context, we chose to represent component assemblies as composite objects. We therefore worked on the reuse of the specifications of the component behaviors during the specification of the behavior of composite objects. Modeling the behavior of composite objects is not straightforward: it requires solutions adapted to their aggregated and dynamic nature that classical object-oriented methods do not provide. We therefore propose an extension to the UML modeling language allowing to efficiently design the behavior of composite objects.

The remainder of this paper presents, on the one hand, how the design of the behavior of a composite object can be achieved with statecharts in UML and how this falls short in providing all the benefits expected from the use of composite objects to model component assemblies. On the other hand, it describes how the semantics of classical statecharts can be refined and extended to provide a dedicated support for the design of composite object behavior.

2. Reusing behavior designs

2-1. Composite objects: an adapted structure for the modeling of component assemblies

Assembling components consists in tying together a set of isolated components so that they can interact and constitute, as a whole, a system which implements the required functionalities. Two kinds of approaches exist for building component assemblies [5]:

- approaches based on connectors [7, 14], which model an assembly of components as simple interconnections between them, without any representation of the assembly itself as an explicit entity. Each assembly is unique and entirely built by the architect.
- approaches based on complex components [12, 24, 27, 28], which model a component assembly as a full fledged object. An assembly of component is then represented in a generic way by an object class. It can be thus reproduced many times and automatically built as an instance of this class.

The concrete representation of a component assembly provide approaches based on complex components with better qualities, regarding reuse, than its connector-based counterpart.

In our work, the modeling of component assemblies as complex components is based on composite objects. A composite object is an object which represents the aggregation of a set of objects, called its components, each of which describes a part of the composite object [3]. A composite object embodies, as an abstract entity (a unique object), the assembly built with its components. Using composite objects to represent component assemblies provide incontestable benefits:

- **richness and expressivity of the modeling of assemblies**. The structure that links a composite object to its components is explicitly represented by a set of relations, called composition relations [11, 19, 22]. Most of the existing work on composite objects focuses on the semantics of composition relations in order to use composite objects as a mean to model and manage complex data structures. The richness of this semantic is profitable to the management of component assemblies, in that it allows to control the dynamic evolution of their structure — addition or removal of components — or the access to individual components — by modeling their visiblility or sharability.
- **enhanced reuse thanks to the abstract and hierarchical modeling of assemblies**. Composite objects which model complex entities are designed in a top-down way, by decomposing them, gradually, into a set of smaller parts. On the one hand, this enables each of these parts to become a potential component which can itself be reused for the modeling of other composite objects. On the other hand, it allows the reuse of existing components for the modeling of some parts of the composite object. Moreover, complex assemblies, once represented as a unique abstract entity by a composite object, can be easily reused as components for building larger assemblies in a bottom-up way [3, 24, 27, 28].

Figure 1 depicts such a composite object class, *HeatingSystem*, which models electric heating installations — as for a building CAD application. A heating system is composed of radiators, of a clock and of two thermostats. These thermostats, situated in some central place, allow the ambient temperature of the building to be controlled according to a day / night cycle determined by the clock. A radiator is itself a composite object composed of a heating element and a switch, allowing the radiator to be turned on and off. A radiator can optionally have a thermostat which allows the temperature of the room to be individually set.

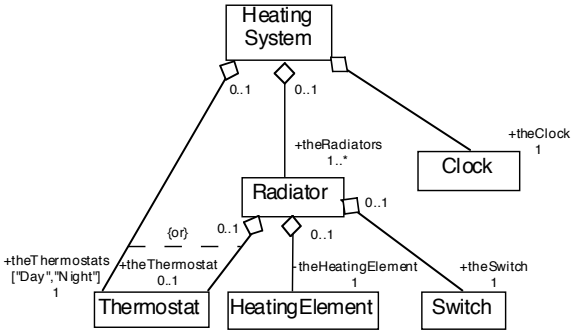

Figure 1 : Example of a composite class modeling heating systems

2-2. Modeling the behavior of composite objects

Whereas a lot of research work studies composite objects from a structural point of view, the behavior of composite objects gets little attention. Yet, modeling the behavior of composite objects requires adapted means to achieve contradictory goals: design behavior of the composite object as a whole, by combining the individual behaviors of its components, while preserving the reusability of each component. Models used to design the behavior of composite objects should have two fundamental qualities:

- **a low coupling of components**, in order to be able to reuse them in different contexts. The contextual information related to the use of a component in a particular composite object must be located in the composite object itself, as a natural representation of the context. This principle maintains the independence of components [3, 21] by preventing specific information from polluting their generic definition and diminish their reuse potential.
- **an abstract definition of the behavior** of the assembly. From a structural point of view, a composite object represents a component assembly as a single abstract entity, making its management and reuse easier. In other words, building a composite object results in creating a higher abstraction level which enables the complexity of object structures to be reduced. Such an abstraction mechanism must be provided for behavior as it already is for structures, in order to express the behavior of composite objects as simply

as possible and make the reuse of composite object behavior designs effective, when using them to build hierarchically higher composite objects.

With respect to this analysis, the following paragraph studies how classical object-oriented design methods manage the modeling of composite object behavior.

2-3. Composite object behavior modeling capabilities of classical object-oriented design methods

2-3-1. Low coupling of components

Classical object-oriented design methods, such as those using the UML standard modeling language, do not provide any specific concept for the design of the behavior of composite objects. The behavior of a composite object must therefore be designed by the means of a simple set of cooperating objects. In this context, the main task of the architect is to choose a cooperation pattern that allows composite objects to control the behavior of their components (execution of functionalities, state changes) in order to define and manage the behavior of the whole. These cooperation patterns are not specific to composite objects nor to a given design method or language. All of them — except event notification — can be represented, to some extent, in UML, thanks to statecharts. The most noteworthy cooperation patterns are:

- **delegation**. This pattern consists in designing a component in such a way that it invokes determined functionalities of the composite objet to which it belongs in order to delegate to it the handling of the global side effects of the execution of its own functionalities. This simple pattern entails that components know the exact type of composite objects to which they can belong, because of the explicit invocation of some of their functionalities. Such contextual information pollutes the generic definition of a component and prevents him from being reused by other unforeseen types of composite objects.
- **event notification** (presented in [10] as the Observer pattern). This popular pattern consists in designing a component in such a way that it notifies observers — in this case, the composite objects to which it belongs — of interesting situations — such as state changes — by firing abstract events. Thus, components do not need to have any knowledge about how composite objects will manage their behavior. This strongly isolates the design of components from the design of composite objects. However, component designers must foresee what will be the possible uses of this component in order to design a set of events that will enable the management of the component. This can result in limitations in the reuse of components if this set of events is not suitable for a specific, unforeseen role of the component in a potential composite object.
- **master-slave model**. This pattern consists in encapsulating all the components of a composite objet inside it. The composite object then has a total control of the behavior of its components (its is the only object which can send messages to them) and does not need to be warned of what they are doing, to manage the behavior of the whole, anymore. Thus, components can have context-free designs, like isolated objects' have. Nonetheless, a master-slave model hampers some qualities of the object structures modeled with composite objets. As components are no more visible, the double view of a composite object, both as an abstract entity and as a detailed structure is no more available. The dynamic, hierarchical structure of a composite object is replaced and poorly reflected by the flat, monolithic and static amount of information provided by the sole composite object interface [3]. Moreover, component sharing is no more possible.
- - **active behaviors**. This pattern consist in designing the behavior of a composite object in such a way that it monitors a set of special situations about its components to which it must react in order to manage its global behavior. To our opinion, this pattern is the most adapted to the design of the behavior of composite objects, for it provides a way to specify how the integration of the behavior of a component is managed, without any requirement in the design of this component, while preserving the component visibility. Such a design of the behavior of a *Radiator* composite object is presented on Figure 2. Active behaviors are expressed thanks to change events on transitions [29] which depicts how a *Radiator* composite object monitors certain state changes in its components and reacts to their occurrences by provoking subsequent state changes to other components in order to combine their behaviors and obtain a global coherent state. For example, when it is in the state *SwitchOff,* a *Radiator* composite object monitors the state of its *Switch* component in order to detect when it is switched on — as specified by the change event defined by the label *when theSwitch in On* on the outgoing transition. When this state change occurs, the transition fires and may result in a coordinated state change of the *HeatingElement* component, depending on the state of the *Thermostat* component, thanks to the message sending specified in the actions of the transition.

The above discussion shows that a low level of coupling can be reached in the design of composite object behaviors, provided an adapted cooperation pattern is chosen to manage the interactions between the composite object and its components, such as active behaviors.

2-3-2. Abstracting the behavior of a composite object

Whereas coupling is a general issue in object-oriented design that can be solved by applying existing pattern with standard notations, abstraction is a specific property of composite objects that distinguish them from flat object systems. Due to the lack of specific notations or concepts, the behavior of a composite object must be designed as the behavior of a system of objects. Thus, the statechart of a composite object is described as a mere juxtaposition of its own statechart to the statecharts of its components (see Figure 2). At each time, the state of the composite object is defined as the product of the active state in each of these diagrams. Coherent combinations of states are obtained by specifying how state changes are synchronized by sending messages between objects. The actual states of a composite object are never explicitly defined: they are calculated by simulating the running of the statemachines into which the statecharts map (through message sendings and transition firings). Designing the behavior of a composite object can thus be compared to designing a complex state, like the *Live* state in Figure 2, by decomposing it into different concurrent regions. However, unlike the states of a single statechart, different statecharts, describing the behavior of different objects, cannot be composed into a single one.

Such behavior specification does not provide any kind of abstraction — i.e. complexity reduction — of the behavior of a composite object. It cannot be reused to design a more complex object without leading to an unmanageable explosion in the number of states and coordinated state change predictions. Moreover, there is no suitable solution to managing the dynamic structure of composite objects (components being added or removed) and its impact on its behavior. The only practical solution is to provide a separate state diagram for each different configuration of a composite object. It results in a set of behavior diagrams for which no formal structure or management concepts are provided.
These important limitations of the modeling languages of classical object-oriented methods, regarding composite object behavior abstraction, motivate the proposition for better adapted solutions. In the following section, we present COBALT, which is our solution to this problem.

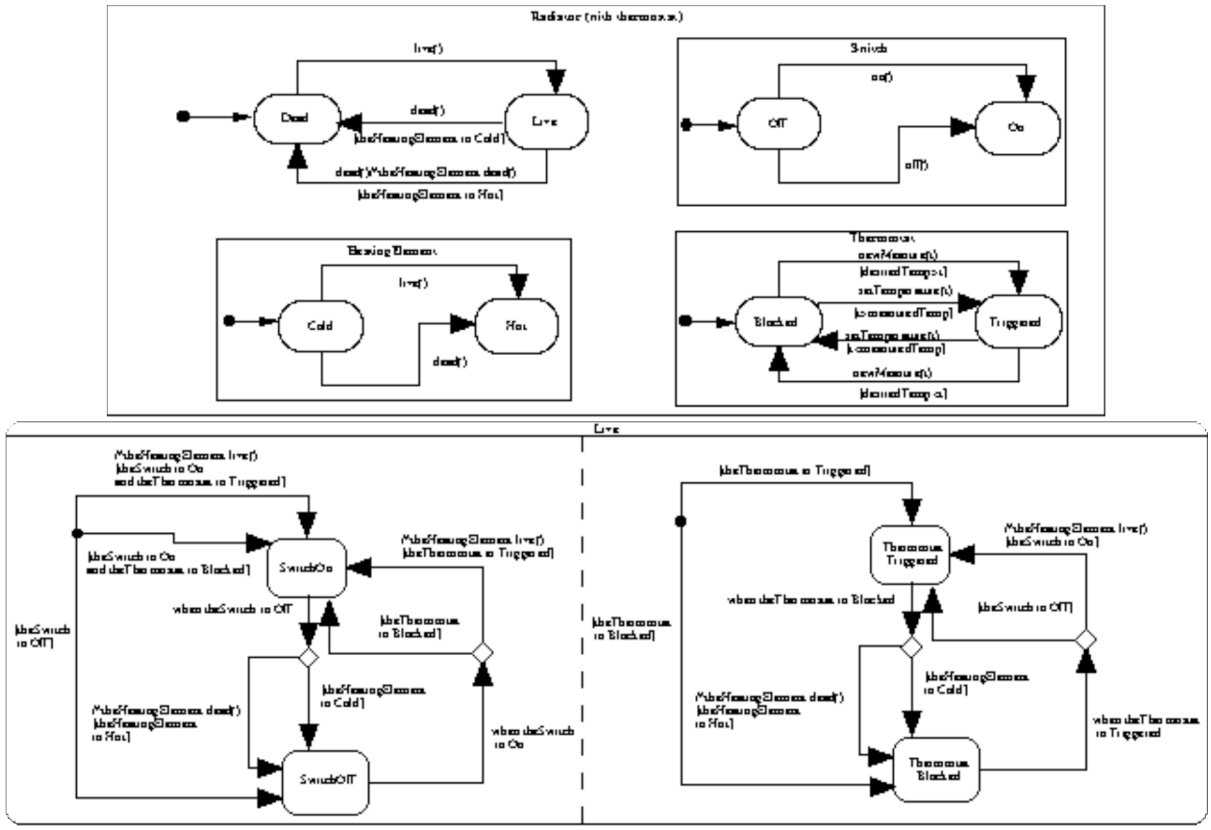

Figure 2 : Statechart diagram of the composite class *Radiator*

3. Another approach for modeling the behavior of component assemblies with statecharts.

As in CATALYSIS [7], we worked on a specific component-based development process, called COBALT [31, 32], which introduces modifications and extensions to the UML notation in order to provide an efficient support for designing and reusing components. But, whereas CATALYSIS promotes a connector-based approach, in which inter-object collaborations are modeled as first-class, reusable design elements, COBALT promotes a complex-component-based approach in which component assemblies are explicitly modeled by composite objects. For this purpose, COBALT uses a variant of statechart diagrams whose semantics is adapted to the modeling of composite object behavior.

3-1. Statechart semantics and notation in COBALT

UML, as a generic modeling language, proposes a rich notation for describing statecharts, which provides them with a versatile semantics that can be adapted to many kinds of behavior designs [31]. On the contrary, COBALT's statecharts have a precise semantics and a single purpose: designing the dynamic behavior of objects — i.e. the set of states in which an object can be and the set of transitions that are possible between these states, defining the different possible life-cycles of this object. Figure 3 presents the statechart diagram of the composite class *Radiator*, as defined in COBALT with respect to the principles defined afterwards.

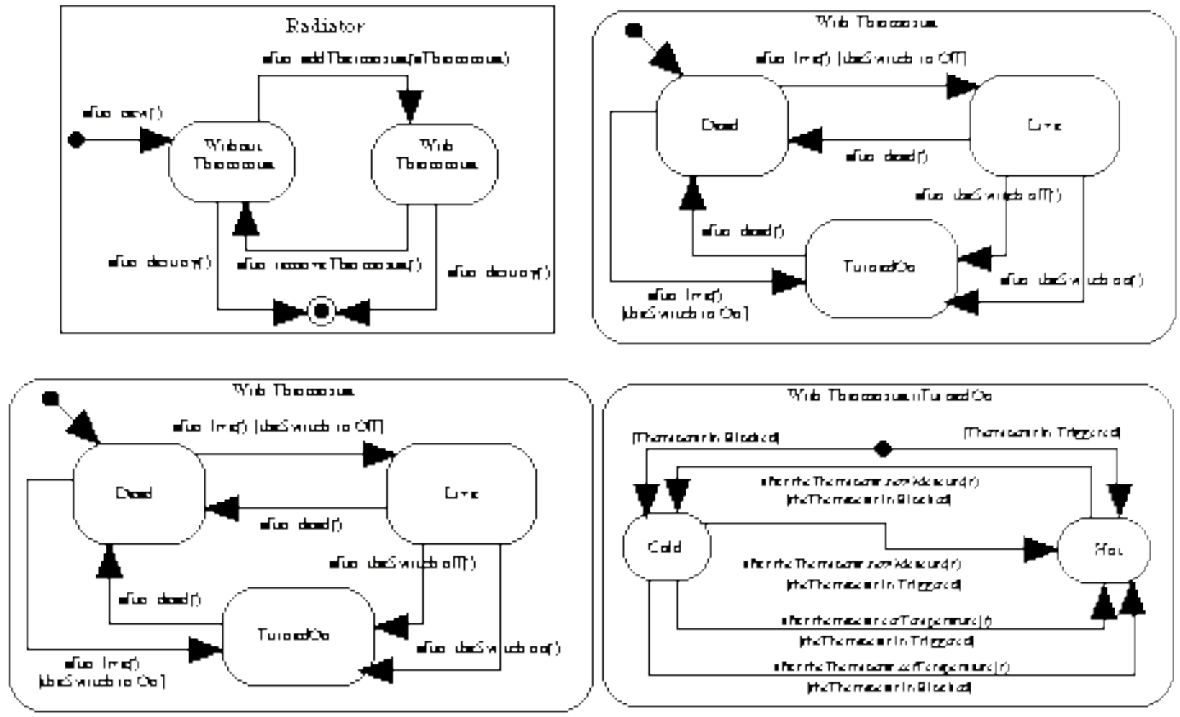

Figure 3 : Statechart diagram of the composite class *Radiator* in COBALT

In COBALT, as in some other work [6, 1, 26], states represent the different behavior modes of an object. Each state models a set of concrete values of the object for which it has some specific behavior. The set of the states of an object is therefore partitioning its different values with regards to the different distinct behaviors they condition. A set is thus only representing an abstract set of concrete values of an object and not the execution of any activities or actions (no use of activities or actions in the description of states). In the same way, a transition only represents the evolution of the state of an object as a result of the execution of a functionality. Transitions do not carry any functional information describing how functionalities are executed (no more actions are associated with transitions). Only the functionalities that really modify the state of the object are relevant for the definition of the statechart (no more internal transitions). Therefore, a statechart does not specify the protocol of an object any more (the sequences of function calls which are valid for the object). In COBALT, the protocol is defined on the functional view of the behavior of an object [31, 32] — on which the detailed execution of each functionality is specified — in order to make statechart specification simpler and to keep dynamic and functional behavior specifications separated. Transitions are triggered by a unique kind of events that signals the end of the execution of functionalities. These events are defined by the keyword *after* (initially used in UML to signal temporal events) followed by the signature of the functionality. The guard which is associated with the transition

is used to measure the effects of the execution of the functionality on the concrete state of the object (the values of its attributes) in order to determine if a transition was crossed and a new abstract state reached.

The statechart diagram of a composite object models the global behavior of the component assembly that it represents. Each state of this diagram is an abstraction of a set of values of the composite object and its components that identifies a specific way the composite object behaves as a whole. Thus, the state of a composite object evolves as defined by its statechart diagram, not only when the own value of the composite object is modified, but also when the value of one of its components is modified. In this letter case, the state change of the composite object is modeled by a transition triggered by the end of the execution of a functionality of a component. This event is specified by prefixing the signature of the executed functionality with a composition path that uniquely identifies the concerned component [3].

3-2. Hierarchical statecharts of composite objects.

As in UML, COBALT's statecharts are hierarchic : they allow statecharts to be constructed by refining states into sets of nested sub-states. When designing the behavior of a composite object, this feature is used in a specific way in order to express the special kind of state changes that come from the dynamic structure evolution of a composite object, in order to specify how the behavior of a composite object is modified when a component is added or removed. To this matter, states at the highest nesting level are given a special role : they model the different kind of configurations of the composite object, meaning the different sets of components for which the composite object has notably different behaviors. For instance, the statechart diagram of the composite class *Radiator* is defined at its highest level by the *WithoutThermostat* and *WithThermostat* states (cf. Figure 3).
Such states are called the *configuration-states* of the composite object. They are linked by transitions triggered by the execution of the functionalities which carry out structural manipulations (addition or removal of a component) to get from a remarkable configuration to another remarkable configuration. The configuration which is associated with a configuration-state can be explicitly defined in a *configuration diagram.* A configuration diagram is drawn as a zoomed state which contains an object diagram depicting the configuration of the composite object in this state. For instance, the configuration diagram on Figure 4 describes the configuration of a *Radiator* composite object when it is in the *WithoutThermostat* configuration-state.

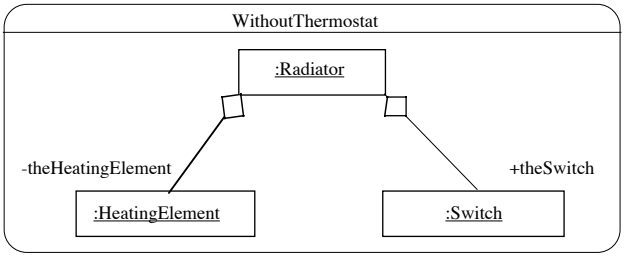

Figure 4 : Configuration diagram associated with the *WithoutThermostat* configuration-state

Each configuration-state is further decomposed in a classical way, by nesting statecharts which describe the behavior of the composite object in each configuration (cf. Figure 3).
The configuration-state concept allows statecharts of composite objects to be rationally constructed. The dynamic behavior of a composite object can be expressed as a unique statechart which integrates in a uniform way every kind of state changes that a composite object may experience. Configuration changes are stable elements in the evolution of the state of a composite object, in comparison with evolutions entailed by value changes of a composite object and its components. Moreover, a configuration change generally entails a more global and drastic variation in the behavior of a composite object than a simple value change. Thus, configuration-states are ideally positioned at the highest abstraction level of a statechart. They improve the hierarchical organization of the statechart of a composite object by partitioning its behavior in a priori radically distinct categories.

3-3. Composition of a composite object state diagram with those of its components

The notations we have introduced so far do not allow to specify which component state combination is represented by each state from the composite object statechart. For this purpose, COBALT introduces a new type of diagram, named *state hierarchization diagram,* which complements the statechart. A state hierarchization diagram shows a zoomed-in representation of a state of the composite object which describes, in nested boxes, the combination of component states defining a given state of the composite object.

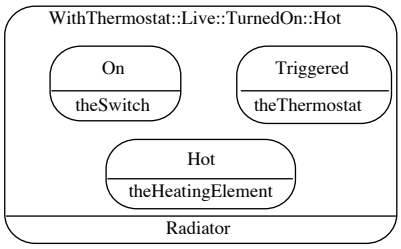

Figure 5 : Specification of the state *WithThermostat::Live::TurnedOn::Hot* of a *Radiator* composite object

For example, the state hierarchization diagram showed on Figure 5 specifies that the state *WithThermostat::Live::TurnedOn::Hot* of an instance of the composite class *Radiator* is obtained by the combination of states *On*, *Triggered* and *Hot* of its *Switch*, *Thermostat* and *HeatingElement* components. This form of nesting is different from the state nesting that can be obtained with the classical statechart hierarchization. Our state hierarchization diagrams do not decompose a given object's state (in this case a given composite object's state) into sub-diagrams containing sub-states of this given object. On the contrary, they express a combination between a state of a composite object and states of other objects (namely, its components).

The most classical combination of component states described by state hierarchization diagrams is the one represented on Figure 5. It consists of the conjunction of a set of states of the composite object's components (AND logical operator). It is represented by the simple juxtaposition of the concerned states. The composite object's components, the state of which does not appear on the state hierarchization diagram, can be in any possible state. We also propose notations allowing to express on state hirarchization diagrams whatever logical combination that can be made with states linked with OR, AND or NOT operators [31, 32]. The main advantage of state hierarchization diagrams is to define the desired state combinations between the composite object and its components :

- **explicitely**. The combinations can directly be read from the diagrams.
- **declaratively**. Neither the statechart diagram nor its associated state hierarchization diagrams concretely define how the combination of states is obtained. It is the responsibility of the functional view of behavior to define the successions of executions of functionalities of the composite object and of its components which cause the desired combinations.

Consequently, state hierarchization diagrams resembles other notions presented in research work that declaratively define state synchronization constraints between a composite object and its components [18, 24]. The set of state hierarchization diagrams associated with the specification of the dynamic behavior of a composite class can be analyzed as a set of state synchronization constraints that must be satisfied during the whole lifecycles of these objects. Despite this similarity, the state hierarchization diagram concept is much more adapted to the semantic specificity of composite objects than its generic counterpart. Unlike a simple constraint between the states of a set of related objects, a state hierarchization diagram defines how a composite object's state is decomposed into a combination of its components' states. State hierarchization diagrams therefore establish a composition hierarchy among states parallel to the composition hierarchy among objects. It provides designers with an abstraction mechanism that is adapted for modeling the behavior of composite objects.

3-4. Usage and benefits of COBALT's statecharts

One of the main goals of COBALT is to allow the specifications of the behavior of a composite object to be constructed by composing the specifications of its components, as structural notations allow a composite object to be described as being composed of other objects. The specific semantics and notations that were introduced above for statecharts achieve this goal for specifying the dynamic behavior of composite objects[1]. The dynamic behavior of a composite object can be expressed on a unique statechart diagram partly composed of elements from the statecharts of its components. Indeed, the states of a composite object are defined as composed of states of its components. In the same way, the set of transitions between these states is partly composed of transitions triggered by the execution of functionalities of components which mirrors actual transitions in the statecharts of these components. The abstraction capabilities of composite objects are thus preserved on the behavioral view of their specifications. The behavior specification of composite objects can therefore be profitably used to manage the reuse process of composite objects. The increasing complexity of the objects that are built through successive

[1] Besides, we also provide in COBALT such extensions for specifying the functional behavior of composite objects, in order to allow the functionalities of a composite object to be defined as a dynamic and declarative composition of the functionalities of its components [31, 32].

compositions is then balanced by an increased abstraction level, which keeps the specifications of composite objects simple and usable.

Besides, in classical object-oriented design methods, statecharts are mainly constructed bottom-up, to sum up the scattered functional behaviors contained in the interaction diagrams resulting from the refinement of use cases. To our opinion, the design process of a component should be inversely carried out — i.e. mainly in a top-down way — because a component is not designed to satisfy the specific requirements of an application but to generically model a concept. This reflection about the intrinsic nature of a component need to be captured on a global and abstract view of its behavior which provide a starting point to be refined into complete detailed behavior specifications. The modal and declarative semantics of our statecharts are suitable for this use. They provide simple, yet comprehensive and formal view of the behavior evolution of an object over its different life-cycles. Once established, a statechart is used as a basis to design the detailed execution of its functionnalities. Indeed, the set of states on a statechart draws a synthetic view of the different situations to be handled by distinctive behaviors of the functionalities (modal behaviors). In the same way, the transitions of a statechart establish a cause-to-effect link between the execution of given functionalities and state changes which provide designers with pre-conditions (source states) and post-conditions (target states) that can help the design of functionalities and allow their specifications to be checked.

The precise semantics of our statecharts allows their interpretation to be fully automated [31]. Beyond their prevalent role in the conceptual steps of component development, statecharts can thus also be used in a concrete way by CASE tools to simulate the behaviors of object assemblies upon their specification, in order to verify their consistency. In the same way, statecharts can be used during the coding phase to automatically generate the code that manages the state changes of objects and the different modes of their functionalities (thanks to a metaobject protocol implementing an extension of the STATE pattern [10,31]).

Once a component is designed and implemented, its statecharts plays a preponderant role in its selection for reuse. The role of a statechart for the dynamic behavior of an object can be compared to the role of the interface for the functional behavior of this object. It provides designers with an abstract view of the dynamic behavior of an object that enables its conformance to requirements to be finely analyzed [13]. Indeed, pure syntactic conformance — i.e. the matching of the signatures of functionalities — cannot ensure more than a “mechanical” compatibility. Statecharts convey enough semantics to check if a component provides a really conformant behavior — i.e. if a given sequence of function calls entails a given sequence of states so that the component can meet the state combinations required to be coherently used in the assembly. After the component is selected, its statechart is used to compose the statechart of the composite object that models the assembly. Then, the statechart of the composite object will be used as a basis for the design of the detailed behavior of the assembly and finally for the automatic code generation the code that implements the management of its behavior.

4. Conclusion and perspectives

This paper presents some of the main features of the COBALT component-based development method. This method lies on the representation of component assemblies as composite objects. In COBALT, we proposes a solution to overcome the lack of abstraction of UML statechart diagrams in order to be able to veritably compose the statechart diagram of a composite object from those of its components. The dynamic behavior definition of a component assembly can therefore benefit from the composite object concept in terms of modeling, management and reuse, in the same way as its structure does. We therefore believe that statechart diagrams, as they are specified in COBALT, can play a central role, not only in the design but also in the coding of a component assembly. For this purpose, we also have proposed an implementation model that enables a direct mapping of a graphic behavior specification into a coding structure that can be implemented into whatever classical object-oriented language. As a perspective and inspired by the JAVABEANS component model [30], we plan to further explore these concepts in order to provide designers with a conceptual view of component behaviors from which he/she can intuitively and declaratively build most of the behavior of an assembly. This conceptual view, described in the JAVABEANS model by the set of events components can emit, would be, in our case, described by the component statechart diagrams.

5. References

[1] P. ANDRE, F. BARBIER, J.C. ROYER, *Une Expérimentation de Développement Formel à Objets,* Technique des Sciences Informatiques, vol. 8, n°14, 1995.
[2] APPLE COMPUTER INC., *WebObjects Developer’s Guide*, 1998.
[3] E. BLAKE, S. COOK, *On Including Part Hierarchies in Object-Oriented Languages, with an implementation in Smalltalk*, ECOOP'87 Proceedings, 1987.

[4] M. BOUZEGHOUB, G. GARDARIN, P. VALDURIEZ, *Les Objets*, Eyrolles, 1997.
[5] C. CAUVET, F. SEMMAK, *La réutilisation dans l'ingénierie des systèmes d'information*, Génie Objet – Analyse et conception de l'évolution, C. OUSSALAH et alii, Hermès, 1999.
[6] P. COAD, E. YOURDON, *Object-Oriented Design*, Yourdon Press / Prentice-Hall, 1991.
[7] D. F. D'SOUZA, A. C. WILLS, *Objects, Components and Frameworks with UML – The Catalysis Approach*, Addison-Wesley, 1998.
[8] M. FOWLER, *Analysis Patterns : Reusable Object Models*, Addison Wesley, 1997.
[9] A. FRONT-CONTE, J.P. GIRAUDIN, D. RIEU, C. SAINT-MARCEL, *Réutilisation et patrons d'ingénierie,* Génie Objet – Analyse et conception de l'évolution, C. OUSSALAH et alii, Hermès, 1999.
[10] E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES, *Design Patterns – Catalogue de Modèles de Conception Réutilisables*, International Thomson Publishing France, 1996.
[11] M. HALPER, J. GELLER, Y. PERL, *An OODB Part Relationship Model*, CIKM'92, 1992.
[12] T. HARTMANN, G. SAAKE, R. JUNGCLAUS, P. HARTEL, J. KUSH, *Revised Version of the Modelling Language TROLL*, tech report 94-03, Technische Universität Braunschweig, April 1994.
[13] D. HAREL, E. GERY, *Executable Object Modeling with Statecharts*, IEEE Computer, February 1997.
[14] R. HELM, I. HOLLAND, D. GANGOPADHYAY, *Contracts : specifying behavioral composition in object-oriented systems*, OOPSLA'90, 1990.
[15] INPRISE CORPORATION, *Building Applications with JBuilder*, 1999.
[16] J. JACOBSON, M. CHRISTENSON, P. JONSSON, G. OVERGAARD, *Object-Oriented Software Engineeting*, ACM Press, 1992.
[17] I. JACOBSON, G. BOOCH, J. RUMBAUGH, *The Unified Software Development Process*, Addison-Wesley, 1999.
[18] G. KAPPEL, M. SCHREFL, *Object/Behavior Diagrams*, IEEE International Conference on Data Engineering '91 Proceedings, pages 530-539, 1991.
[19] W. KIM, *Composite Objects Revisited*, ACM SIGMOD International Conference on Management of Data, 1989.
[20] P. MONDAY, J. CAREY, M. DANGLER, *San Francisco Component Framework : an Introduction*, Addison-Wesley, November 1999.
[21] M. MURATA and K. KUSUMOTO, *Daemons: Another Way of Invoking Methods*, Journal of Object-Oriented Programming, vol. 2, n°2, 1989.
[22] J. ODELL, *Six different kinds of composition*, Journal of Object-Oriented Programming, Jan-Feb 1944.
[23] OBJECT MANAGEMENT GROUP INC., *OMG Unified Modeling Language Specification version 1.3 alpha R5*, March 1999.
[24] D. RAMAZANI, G. von BOCHMANN, *Extending OMT for the Specification of Composite Objects*, TOOLS USA'96 Proceedings, 1996.
[25] REAL SOFTWARE INC., *RealBasic 2.0 Developer's Guide*, 1998.
[26] C. ROLLAND, *Conception de bases de données : une méthode orientée objet et événement*, Technique de l'ingénieur, traité d'informatique, cahier H3248, 1996.
[27] J. RUMBAUGH, *Building boxes : Composite objects*, Journal of Object-Oriented Programming, Nov-Dec 94.
[28] J. RUMBAUGH, *Taking things in context : Using composite objects to build models*, Journal of Object-Oriented Programming, Nov-Dec 95.
[29] SUN MICROSYSTEMS INC., Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition, March 2000.
[30] SUN MICROSYSTEMS INC., Java Beans API Specification 1.0.1, July 1997.
[31] S. VAUTTIER, M. MAGNAN, C. OUSSALAH, *Extended Specification of Composite Objects in UML*, Journal of Object-Oriented Programming, May 1999.
[32] S. VAUTTIER, *Une Etude de la Modélisation du Comportement des Objets Composites*, Thèse de Doctorat de l'Univesité de Montpellier II, Décembre 1999.