Can Intuition Become Rigorous?
Foundations for UML Model Verification Tools[1]

José Luis Fernández Alemán, Ambrosio Toval Álvarez
Software Engineering Research Group,
Department of Informatics and Systems, University of Murcia (Spain)
{aleman, atoval}@dif.um.es

Abstract

The Unified Modeling Language, UML, is the object-oriented notation adopted as the standard for object-oriented Analysis and Design by the Object Management Group. This paper reports on research to facilitate the formal revision of UML informal specifications. The approach is based on the algebraic specification formal theory, which is used to formalize the UML Statechart Diagrams and subsequently verify them. To illustrate the proposal, the so-called orthogonality property is investigated. This property is modeled at the UML metamodel level so that its fulfillment on the part of any particular UML Statechart diagram can be mathematically proven or disproven. The formal models obtained are specified in the executable formal language Maude thus providing the additional advantage of using them as functional prototypes. These results lead up to a whole formalization of the UML, which can be used in practice, and lay the foundations for the construction of rigorous UML CASE tools.
Keywords: Requirements Specification, Formal Verification, UML Statechart Diagrams, Software Model Reliability.

1: Introduction

It is widely accepted by the Software Engineering community that functional requirements verification through revision of the different models from the early stages of system development life-cycle is critical in producing reliable software systems. Typically, these models are constructed by means of some graphical notation. However, the imprecise semantics of most current OO methodologies and graphical techniques often leads users and analysts to ambiguous interpretations, causing errors in the early stages of the life-cycle that can be critical for the rest of the development. To quote Harel [19], unless these pictorial notations are backed up by precise semantics, they remain only pretty pictures. An important goal is to combine the intuitive appeal of visual notations with the precision of formal specification languages.

Despite their negative aspects (e.g. learning difficulties or lack of efficient tools), Formal Methods (FM) can play an important role to cope with this issue. The formalization of a graphical notation helps to identify and remove these ambiguities. FM also provide other advantages: improving the graphical notation (by detecting ambiguity and imprecision), or improving the formal language used (as a consequence of trying out the expressiveness of the language); it also facilitates a rigorous maintenance of cross-references between models, the automatic generation of prototypes and the demonstration of properties. This paper reports on research to facilitate the formal revision of UML informal specifications.

The Unified Modeling Language, UML, is the object-oriented notation adopted by the Object Management Group as the standard. Despite this fact, and as it has been recognized by its authors, the current UML semantics is not still sufficiently precise. For example, the UML static semantics is described by a semi-formal language, the Object Constraint Language (OCL), and the UML dynamic semantics is expressed in informal English. Recently, Reggio and Wieringa [29]

identified and reported thirty one particular problems in the current UML dynamic semantics.

Our current research on this topic has produced formal models for the UML Statechart Diagrams and Class Diagrams, and OCL. Taking into account that state machine formalisms predominate as a conceptual framework, as it has been reported in a systematic survey and analysis of the use of FM in the Industry [7], we have chosen the UML Statechart Diagrams to describe our UML formalizing approach. Therefore, in this paper a proposal to formalize the UML Statechart Diagram syntax and static semantics is presented. This kind of diagram is not free from the UML semantic problems mentioned above. In the UML documentation a number of constraints are given using OCL to express correct transitions. However, it is not mentioned, for instance, that the set of source states (similarly target states) of a complex transition should be mutually orthogonal. Likewise, the UML definition does not cope with the orthogonality constraints of compound transitions with multiple source states and multiple destination states at the same time. These are some of the issues considered here.

The approach followed in this paper is based on the formalization of the UML metamodel [24][25], thus obtaining the necessary conceptual framework to specify domain models. The UML syntax is described by the so-called *syntactic specifications*, incorporating the static semantics by means of equations or axioms. Hence, any UML Statechart diagram describing a particular system (or a part of it) can be translated to its equivalent formal representation, which in our case becomes a formal term of the specified algebra. The current formal specification provided in this paper does not cope with the UML Statechart Diagram dynamic semantics. This will be the subject of future work. We have put the current formal models into practice by integrating them into an OO prototyping and verification process model [23] that combines rapid and evolutionary prototyping strategies, to obtain EIFFEL programs.

The rest of the paper is organized as follows: section 2 briefly describes the UML Statechart Diagrams notation. Section 3 outlines the formal techniques used, namely the algebraic specification theory and the Maude language. Afterwards, in the same section, a formal framework for rigorously denoting and verifying the UML Statechart Diagrams is given, in the context of the whole UML. Section 4 discusses a procedure for the application of such framework to the verification of particular properties on the UML Statechart Diagrams. Section 5 compares our formalization with related work. Finally, section 6

presents some concluding remarks and the work to be done in the future.

2: The UML Statechart Diagrams

The UML Statechart Diagrams [24] [25] are substantially David Harel's Statecharts [18] extended with OO characteristics. A UML Statechart diagram can be seen as a graph of states connected by labeled transitions. In UML, a state can be simple or composite. A composite state consists of either concurrent substates or disjoint substates. A state (as defined by the UML glossary) denotes a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. OR (actually XOR) composite states are classical states of state machines (figure 1a), that is, an object cannot be found in two substates of an OR composite state at the same time. In contrast, AND composite states, as introduced by Statecharts, offer the possibility of representing concurrent behavior, and provide a state hierarchy that reduces the complexity of the diagram (figure 1a). These diagrams are mainly used to represent behavior where asynchronous events are involved.

In section 4, an alternative notation that helps understand the orthogonality (i.e. concurrency) property checking process is used. The new representation of figure 1a is shown in figure 1b.
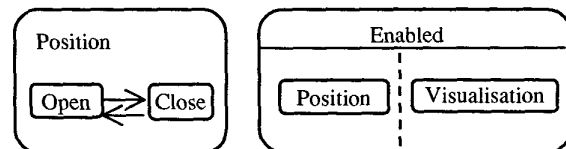

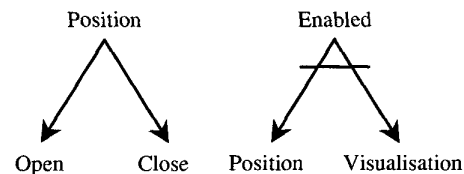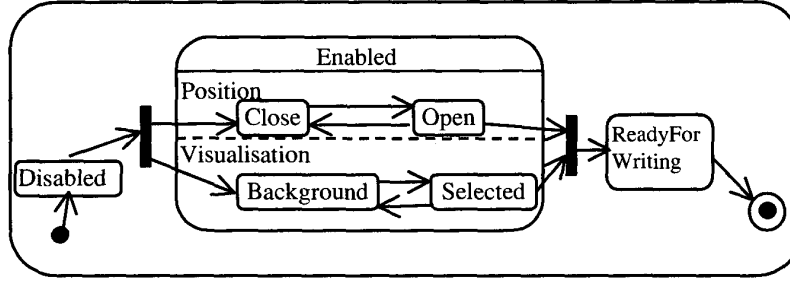**Fig. 1a. OR and AND composite states**

Fig. 1b. OR and AND composite states

Fig. 2. Complex transitions in an AND composite state

An object that remains in a state may be performing some activity, that is, executing an ongoing action, or waiting for some event. An *action state* is a state in which an activity is executed. An action is an abstraction of a computational procedure, which results in a change in the state of the model. It becomes apparent by means of sending a message to an object or modifying a value of an attribute. An action cannot be interrupted.

A transition may be simple or complex. A simple transition has a unique source state and a unique target state, whereas a complex transition may have more than one source state and/or more than one target state. A transition label consists of a name of an event and its parameters, a guard condition or necessary condition to trigger the transition, and some actions executed only if the transition is triggered. Figure 2 shows two simplified examples of complex transition; the first one has a source state, *Disabled*, and two target states, *Close* and *Background*. The other complex transition has two source states, *Open* and *Selected*, and one target state, *ReadyForWriting*.

A UML Statechart diagram can be used to specify behavior of entities such as class instances or to define the interactions between entities such as collaborations. In the following sections only the UML Statechart Diagram features that are relevant to prove the orthogonality property, are considered.

3: Formalizing the notation

In this section we firstly present the formalism used to represent the UML Statechart Diagrams, namely order-sorted algebraic specification theory, and justify the choice. The remainder of the section details a particular strategy for formalizing the notation and shows a small part of the formal specification.

3.1: Order-sorted algebraic specification theory

As a help in understanding the remaining sections, we present some basic notions regarding the formalism used. A more detailed description can be found in any book related to the topic [15] [16]. An OSA signature Σ is a tuple $(S, OP, \leq)$ where OP denotes a set of operation symbols and $(S, \leq)$ is a partially ordered set of sorts. Each sort denotes a collection of distinguished and homogenous data items. Let $s, s_1, s_2, ..., s_n$ be sorts, a function symbol can be a constant symbol $f: \rightarrow s$, such as *empty* : -> *Stack* (figure 3) and *initialState:* -> *State* (figure 6), or a function symbol whose arity is n, $f: s_1, s_2, ..., s_n \rightarrow s$, such as *top* : *NeStack* -> *Nat* (figure 3) and *simpleState: StateName InternalTransition* -> *State* (figure 6).

The set of Σ-terms of sort s, denoted T_Σ, is defined by the following two conditions:

- $\forall s \mid s \in S$, if $f \in OP$ and f is a constant symbol $f: \rightarrow s$, then $f \in T_\Sigma$. For example, given the constant symbol *initialState:* -> *State*, then *initialState* is a term and given the constant symbol *empty* : -> *Stack*, then *empty* is a term.
- $\forall n \mid n \in N$, if $f \in OP$ and f is a function symbol, $f: s_1, s_2, ..., s_n \rightarrow s$, with $t_i \in T_{OP.si}$ (terms of sort s_i), then $f(t_1, .., t_n) \in T_\Sigma$. For example, *push(5, empty)* is a term where 5 is a term of sort *Nat*, and *empty* is a term of sort *Stack*. In another example, *simpleState('Open, empty)* is a term where *'Open* is a term of sort *StateName*, and *empty* a term of sort *InternalTransition* (see figure 6).

Given a signature Σ, a model for Σ is named a Σ-algebra (or just an algebra) and consists of:

- For each sort s where $s \in S$, a set I_s, termed the carrier set.
- For each constant symbol $f: \rightarrow s$, a constant $f_s \in I_s$ (the interpretation of f in I_s).

• For each function symbol f: s_1, s_2, ..., $s_n \rightarrow s$, a function f_s: Is_1, Is_2, ..., $Is_n \rightarrow Is$ (the interpretation of f in I_s).

An order-sorted algebraic specification is a pair (Σ, E) where Σ is an OSA signature and E is a set of equations. For example, "top push (X, S) = X", is an equational axiom of the algebraic specification of a stack (figure 3). A Σ-algebra that fulfils the equations is called a SPEC-algebra or model for the algebraic specification. The order-sorted algebraic specification theory has been chosen, among other reasons that we discuss below, because it is a well-founded mathematical theory based upon a sound logic and can be computationally interpreted.

There is a variety of support software tools that enable the execution of the models obtained by means of this formalism. Among these software tools, Maude [22] has been revealed as the most appropriate choice, according to the goals of this paper. Maude[2] has evolved from OBJ3, enriched with more expressiveness and more efficient software tools. It is an executable language that supports parameterized programming, multiple inheritance, and a *large-grain* programming technique, which provides support for the scalability of the specification and appropriately manages the complexity of a system. Some of these characteristics are also shared by the order-sorted algebraic specification theory itself. Therefore, the algebraic formalism will help to obtain an integrated formal model of the whole UML metamodel, modeling all the diagrams and their relationships in the same theory.

```
(fmod STACK-OF-NAT is sorts Stack NeStack .
    subsort NeStack < Stack .
    protecting NAT .
    op empty : -> Stack
    op push : Nat Stack -> NeStack .
    op top_ : NeStack -> Nat .
    op pop_ : NeStack -> Stack.
    var X : Nat.     var S : Stack .
    eq top push (X, S) = X .
    eq pop push (X, S) = X .
endfm)
```

Fig. 3. Algebraic specification of a stack

Simplicity is another characteristic to bear in mind. Starting from a small number of mathematics elements (sort, operation symbol, variable and equation) it is possible to define complex module specifications, which in addition can be seen as typical ADTs, therefore turning out a familiar notation to the software practitioners.

---

[2] Maude interpreter has been available since January 1999 and it is a more powerful language than OBJ3. Currently, Maude is freeware and runs under Linux.

The algebraic specification language Maude is based upon both equational and rewriting logic. Maude's equational logic, called membership equational logic, is an extension of OBJ3's order-sorted equational logic (the sorts can be ordered by a partial order relation), and its operational semantic is based on rewriting logic [21]. Maude functional modules are executed by interpreting the equations of a specification as a left-to-right term rewriting system. This "reduction process" finishes when a term is reached to which no equation can be applied. The resulting term is named the "canonical term" and the term algebra modulo the congruence generated by the rules of deduction is named the "quotient term algebra" and represents the minimal formal interpretation model. Thus, Maude programs are executable order-sorted algebraic specifications and computation becomes an efficient form of equational deduction by rewriting.

3.2: Formalizing the UML Statechart Diagrams, as a part of the UML metamodel

At first glance, the fact of using algebraic specification to represent dynamic structures may seem surprising. If the goal had solely been to formalize the UML Statechart Diagrams, then another formalism, different from the algebraic one, should probably have been chosen (for instance using graphs [14]). However, bearing in mind the subsequent integration of the UML Statechart Diagrams with the rest of the diagrams of the UML metamodel, it is profitable to use the same formalism to formalize all the parts, the whole and their relationships, thus avoiding the "impedance mismatch". In our opinion, the algebraic formalism is quite suitable to achieve this goal. The same methodology that we describe in this paper to formalize the UML Statechart Diagrams can be applied to the other UML diagrams. In this case, we need to obtain the corresponding algebraic specifications for them, modeling the corresponding elements and relationships at the UML metamodel level. However, the procedure to obtain these specifications is the same as the stated one here for the UML Statechart Diagram syntax and semantics. Preliminary versions of the UML Class Diagram (including its dynamic semantics) [12] and of the constraint language OCL are available. However, another more powerful conceptual foundation will be necessary if our aim is to formalize, not only the UML metamodel, but also its evolution. This issue has been discussed in other papers [10][31].

Returning to the UML Statechart Diagram case, and looking at it in depth, three possible relationships between data types and concurrent systems arise [1]:

- Concurrent systems use data structures. An algebraic specification can appropriately specify an abstract data type that represents a data structure.
- The abstract structure of a concurrent system can be described as an abstract data type. An algebraic specification can relevantly specify a abstract data type that represents a concurrent system structure.
- Concurrent systems can be seen as data. An algebraic specification can specify a dynamic abstract data type.

In most cases the first level is applied, inasmuch as algebraic specification is the most appropriate formalism to specify abstract data types. If it was only applied to this level, then another formalism would be required to specify the concurrent structure [4] [32]. In the second level, it is possible to specify the evolution of a concurrent system by means of abstract data types. This approach is used by several authors [3] [30]. The last level is used when functions and processes have to be seen as data, thus causing dynamic abstract data types to arise.

In this paper the second approach is used. Both syntax and static semantics are specified by means of an algebraic formalization. Thus, a model of a certain problem (the so-called domain model), initially represented as a UML Statechart diagram, is represented by a term in Maude. Figure 4 shows the approach followed in which *Term1... Termn*, *Term1... Termm* are

formal terms of the (syntactic or semantic theories) corresponding term algebras or formal interpretation models; *CanonicalTerm1...CanonicalTermm* are formal terms (of the quotient term algebra) that represent UML Statechart diagrams which fulfil the rules defined in the UML static semantics, whereas *IncorrectDiagram* represents those UML Statechart diagrams that do not. For instance, in the top of figure 4, a number of different incorrect UML Statechart diagrams are firstly translated to the same number of corresponding formal terms of the term algebra. Then, as all of them are erroneous, they are converted (or "reduced", in algebraic terminology) to a unique, minimal term with the same semantics; in this case the term *IncorrectDiagram* from the quotient term algebra.

Likewise, our research lays the basis for formalizing the dynamic semantics in such a way that a state is specified as a term, which is contained in a quotient term algebra, and a step in the execution of the system is represented by a step of rewriting. Equations can be defined so as to reduce a term to another canonical term by using term rewriting. This is in line with other related work in which execution semantics of Petri nets and Superposed Automata nets are described by using algebraic specification theory and OBJ [2]. We use term rewriting here in order to check if a given state hierarchy satisfies the rules defined in the UML static semantics.
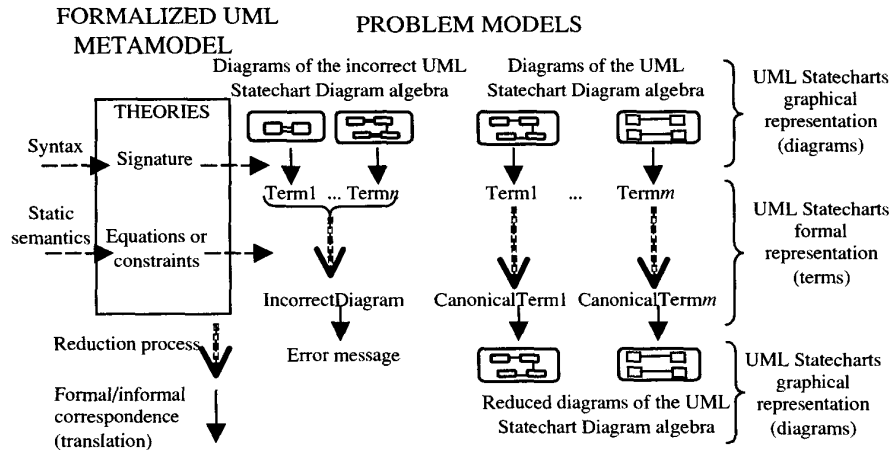

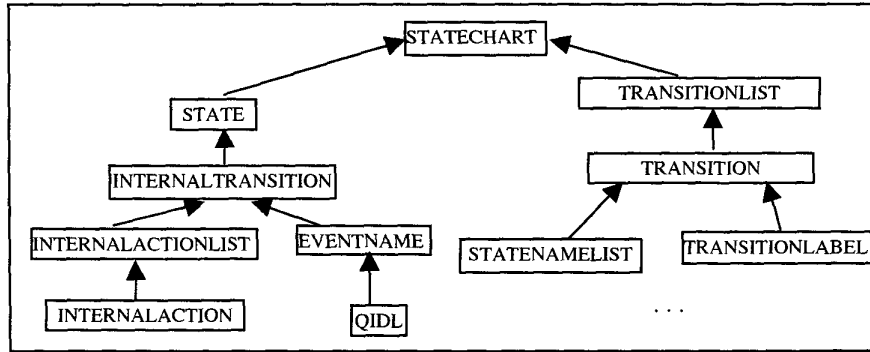**Fig. 4. Formalization of the UML metamodel**

Fig. 5. A Maude module hierarchy for the UML Statechart Diagram algebraic specification

3.2.1: Formalizing the UML Statechart Diagram syntax and static semantics. Before describing the formal specification of orthogonality, the most important Maude modules (i.e. algebraic specifications) of the UML Statechart Diagram formal model are shown in figure 5. They will help us to better understand the whole specification. A UML Statechart diagram is specified in the module *STATECHART* by means of the sort *Statechart*. In this module, equations are defined to check whether a UML Statechart diagram contains duplicated states and whether its state hierarchy satisfies the orthogonality property. The module *STATECHART* imports the modules *STATE* and *TRANSITIONLIST*. The module *STATE* specifies the sort *State* which is used to represent UML Statechart Diagram state hierarchies and provides sorts, operations and equations to deal with lists of states. One of the modules imported by STATE is *INTERNALTRANSITION* which includes the sort *InternalTransition*, denoting activities that are performed while the element is in the state. The appearance of the same internal transition event name more than once per state is checked in *INTERNALTRANSITION*, by using the sorts *EventName*, and *InternalActionList* defined in modules *EVENTNAME* and *INTERNALACTIONLIST* respectively. Both special internal transitions such as *entry*, *exit* and *do*, and general internal actions are declared in the module *INTERNALACTION*. Quoted identifiers and their operations are defined in the built-in module QID.

The module *TRANSITIONLIST* declares the sort TransitionList by importing the parameterized module

LIST. The transitions are defined in the module *TRANSITION* which imports the modules *STATENAMELIST* and *TRANSITIONLABEL*. These modules include the sorts *NonEmptyStateNameList* and *TransitionLabel* which are used to specify a transition.

For the sake of clarity we show only the formalized UML Statechart Diagram features that are necessary to understand and prove the orthogonality property. The interested reader can find all the details of this specification, including the axioms and all of the formalized UML Statechart Diagram features, in a technical document [11]. The formal specifications needed to prove the orthogonality property are addressed in detail below.

The UML Statechart Diagrams are formalized as a piece of the UML metamodel in two steps: firstly, the state hierarchy is formally represented by means of the module *STATE*. Secondly, the list of transitions is formalized by the module *TRANSITIONLIST* (figure 6 shows both signatures). The corresponding quotient algebras satisfy these algebraic specifications (that is to say, they are models, or SPEC-algebras, for *STATE* and *TRANSITIONLIST*), where each item of the carrier set associated with the corresponding data item of sort *State* or *Transition* is a reduced term or normal form deduced by applying the equations. Each item of the carrier set I_{State}, that is to say, the interpretation of each data item of sort *State*, can also be considered as a (real) state hierarchy instead of a term. In the same way, each item of the carrier set $I_{Transition}$ can be considered as a (real) list of transitions instead of a term.

```
(fmod STATECHART is sort Statechart .
  protecting STATE .
  protecting TRANSITIONLIST .
*** operations and equations for property checking
endfm)
(fmod STATE is sort State StateList NonEmptyStateList .
  protecting  ... *** (importing modules)
  subsort State < NonEmptyStateList < StateList .
  op initialState : -> State .
  op finalState : -> State .
  op simpleState : StateName InternalTransition -> State .
  op orState : StateName InternalTransition StateList -> State .
  op andState :StateName StateList -> State .
  op emptyList : -> StateList .
  op errorState : -> State .
  op __ : StateList StateList -> StateList [assoc id:empty] .
  op __ : NonEmptyStateList StateList -> NonEmptyStateList
endfm)
(fmod TRANSITION is sort Transition .
    protecting STATENAMELIST .
    protecting TRANSITIONLABEL .
    op transition : NonEmptyStateNameList NonEmptyStateNameList
                         TransitionLabel -> Transition .
endfm)
(view Transition from TRIV to TRANSITION is
    sort Elt to Transition .
endv)
(fmod TRANSITIONLIST is
  protecting LIST[Transition] * (sort List to TransitionList) .
endfm)
```

Fig. 6. Signature for the states and transitions of the UML Statechart Diagrams

For example, the data item $errorState_{State}$ of the set carrier I_{State} (associated with the constant $errorState$ in module STATE) is a canonical term. It represents the set of the terms (of the term algebra) that denote incorrect state hierarchies according to the rules stated by the UML. The UML Statechart diagram shown in figure 2 is expressed, in terms of this signature, as the pair (*hierarchy, transitions*) made up of the term called *hierarchy* (figure 7a) and the term called *transitions* (figure 7c). They constitute two data items of the carrier sets I_{State} and $I_{TransitionList}$, respectively (that is to say, a "real" and particular UML Statechart diagram, concerning a specific domain). The subterms from $eT1$ to $eT7$ in figure 7c denote the transition labels whose detailed definition is omitted for simplicity. Notice that the sort *Transition* implicitly includes complex and simple transitions. The last ones refers to the case in which the two arguments of the operation *transition* (figure 6) are lists with only one element in each.

On the other hand, an example of a term that represents an incorrect state hierarchy –with a single error- is shown in figure 7b: an OR state called Window has two initial states. This term is reduced by means of term rewriting to the canonical term $errorState_{State}$ by using the Maude "red" command. For simplicity, we introduce the constant term *empty* of the carrier set $I_{InternalTransition}$.

```
hierarchy = orState('Window, empty,
  (initialState
  simpleState('Disabled, empty)
   andState('Enabled,
     orState('Position, empty,
       simpleState('Open, empty)
       simpleState('Close, empty))
     orState('Visualisation, empty,
       simpleState('Background, empty)
       simpleState('Selected, empty)) )
  simpleState('ReadyForWriting, empty)
  finalState)) .
```

Fig. 7a. A correct state hierarchy representing the UML Statechart diagram in figure 2

```
hierarchy2 =
  OrState('Window, empty,
  (initialState
   initialState
   simpleState('Disabled, empty)
   finalState))

red hierarchy2
Result: errorState
```

Fig. 7b. A term representing an incorrect state hierarchy

```
transitions =
    transition ('initialState, 'Disabled, empty)
    transition ('Open, 'Close, eT1)
    transition ('Close, 'Open, eT2)
    transition ('Background, 'Selected, eT3)
    transition ('Selected, 'Background, eT4)
    transition
        ('Selected 'Open, 'ReadyForWriting, eT5)
    transition
        ('Disabled, 'Background 'Close, eT6)
    transition
        ('ReadyForWriting, 'finalState, eT7)
```

Fig. 7c. Transitions of the UML Statechart diagram in figure 2

Term rewriting can be used as a deductive proof system so as to verify properties of a UML Statechart diagram and thus help to increase the reliance in the resulting verified software UML models. In order to illustrate the procedure to achieve this kind of formal verification we will focus on the fulfillment of the orthogonality property for any given UML Statechart diagram.

4: Proving properties: the orthogonality example

Orthogonality is the dual of the OR decomposition of states, being in essence, an AND decomposition [19]. Intuitively, orthogonality requires that being in a state, the system must be in all of its AND components. Therefore, each transition defined in a UML Statechart diagram must preserve the orthogonality defined in the system and fulfil it along the full state hierarchy. In order to suitably formalize this property, we firstly introduce the definition of orthogonal states [18]: let A and B be two states and let us consider the representation of a UML Statechart diagram by means of a state tree (see figures 1b and 11). Then A and B are orthogonal if they are not on the same path and their least common ancestor is an AND state. Detecting the violation of this property permits us to locate and remove errors, which may have critical consequences in real situations. An implementation of an incorrect model where a transition from one state to another must be forbidden would lead to system failures. Likewise, the rewriting techniques help us to rapidly check these kinds of properties in large and complex UML

Statechart diagrams which have many nested states (unlike the diagrams used as examples in this paper).

With this aim in mind, notice that a complex (or simple) transition satisfies the constraints imposed by orthogonality when (see figure 8):

- all of its source states are pairwise orthogonal: to ensure this constraint, the boolean operation *pairwiseOrthogonal* must return *true* when applied to the corresponding term, *pairwiseOrthogonal* (*S, SourceState*)
- all of its target states are pairwise orthogonal: the operation *pairwiseOrthogonal* must return *true* when applied to the corresponding term, *pairwiseOrthogonal* (*S, TargetState*) and
- each target state is non-orthogonal with regard to each source state: the boolean operation *nonOrthogonalSvsT* must return *true* when applied to the corresponding term, *nonOrthogonalSvsT* (*S, SourceState, TargetState*).

S is a variable that denotes the whole state hierarchy; *SourceState* and *TargetState* are variables that denote, respectively, a non-empty list of names of source state and a non-empty list of names of target state, both of them belonging to the same transition. Lastly, the variable *LT* denotes a transition label. In figure 8, the equation that contains the semantics of the orthogonality property is shown. This equation is included in the module *STATECHART*.

```
eq transitionOrthogonal (S,
transition (SourceState, TargetState, LT)) =
 pairwiseOrthogonal (S, SourceState) and
 pairwiseOrthogonal (S, TargetState) and
 nonOrthogonalSvsT (S, SourceState, TargetState) .
```

Fig. 8. Semantics of transition preserving orthogonality

Observe that the UML Statechart diagram in figure 2 fulfils the orthogonality property whereas the one in figure 9 does not. In this case, a complex transition from the state *Disabled* to the states *Close* and *ReadyForWriting* does not satisfy the orthogonality because the target states are non-orthogonal, namely, an object of class *Window* cannot be in both states at the same time.
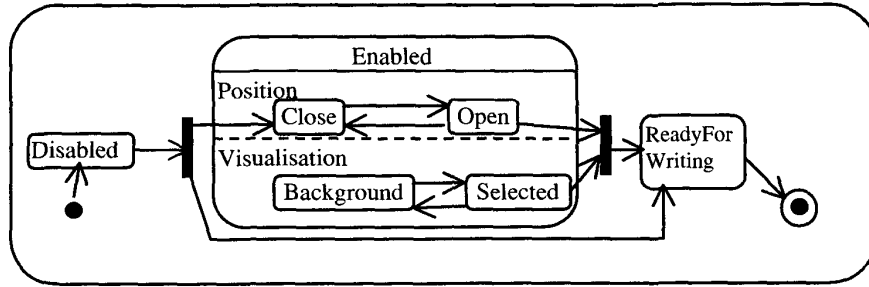
Fig. 9. Complex transition that does not fulfil orthogonality

The term denoting the transitions of the UML Statechart diagram in figure 9 is shown in figure 10. The terms from *eT1* to *eT7* denote the transition labels whose detailed definition is omitted for simplicity.

```
transitions =
  transition ('initialState, 'Disabled, empty)
  transition ('Open, 'Close, eT1)
  transition ('Close, 'Open, eT2)
  transition ('Background, 'Selected, eT3)
  transition ('Selected, 'Background, eT4)
  transition
      ('Selected 'Open, 'ReadyForWriting, eT5)
  transition
      ('Disabled, 'ReadyForWriting 'Close, eT6)
  transition
      ('ReadyForWriting, 'finalState, eT7)
```

Fig. 10. Transitions of the UML Statechart diagram in figure 9

In order to check whether the diagram shown in figure 9 fulfils the orthogonality property, the term in the lefthand side of the equation in figure 8 is instantiated with each transition of the UML Statechart diagram in figure 9 (subterms of the term *transitions* in figure 10). The reduction of this equation instantiated with the last term *transition* ('*Disabled*, '*ReadyForWriting* '*Close*, *eT6*) is shown as follows:

```
red transitionOrthogonal (hierarchy,
    transition ('Disabled, 'ReadyForWriting
                              'Close, eT6))
Result: false
```

where the term *hierarchy* (figure 7a) is bound to the variable *S* of sort *State*, the term '*Disabled* is bound to the variable *SourceState* of sort *NonEmptyStateNameList*, the term '*ReadyForWriting* '*Close* is bound to the variable *TargetState* of sort *NonEmptyStateNameList*, and the term *eT6* is bound to the variable *LT* of sort *TransitionLabel*. The resulting reduction is false because the term *pairwiseOrthogonal* (*hierarchy*, '*ReadyForWriting* '*Close*) yields false.

Regarding the equations for *pairwiseOrthogonal*, in figure 8, the key idea is to know when two given states are orthogonal. For example, the two non-orthogonal states, *ReadyForWriting* and *Close*, are shown in figure 11 by using the notation introduced in section 2. Let us assume that we choose one of the two states, for example *Close*. Let *main path* be the route from the root of the state hierarchy to that state (the gross line in figure 11). All the states of concurrent hierarchies, which branch from the *main path* - *Background* and *Selected-*, are orthogonal to *Close*. Since *ReadyForWriting* is not in the list (*Background Selected*), *ReadyForWriting* is not orthogonal to *Close*.

Figure 12 shows the equations for *pairwiseOrthogonal*. *S* is a variable that denotes the whole state hierarchy; *State* and *StateList* are variables that denote, respectively, a name of state and a non-empty list of names of state. The constant *emptyList* represents an empty list. The equation for *oneToManyOrthogonal* verifies if *State* is orthogonal with regard to each state of *StateList*.
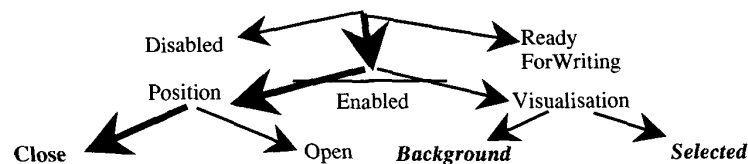

Fig. 11. State hierarchy of the UML Statechart diagram in figure 2 represented by means of a tree

```
eq pairwiseOrthogonal (S, emptyList) = true .
eq pairwiseOrthogonal (S, State) = true .
eq pairwiseOrthogonal (S, State StateList) =
      oneToManyOrthogonal (S, State StateList)
         and pairwiseOrthogonal (S, StateList) .
```

Fig. 12. Equation to verify if the states in a list are pairwise orthogonal

5: Related work

Early attempts were directed towards formalizing OO methodologies such as OMT, Shlaer-Mellor and Fusion, by using LOTOS [4] [32], LCM and TROLL [33], and Z [13], respectively. More recently, some of the UML formal models have been proposed using a variety of formalisms such as graphs [14], mathematical theory of streams and stream processing functions (project SysLab [5]), or the ODAL language, formalized in π-calculus [26]. All the above approaches propose the formalization to verify properties, but, none of them, except that by Wang [32], present any kind of concrete formal verification or example of use.

Pons et al. [27] present an algebraic specification of a data model, similar to the one proposed in UML, so as to specify and verify semantic properties of the specifications. However, its authors do not suggest any method of demonstration nor do they propose the executability of the specification.

The proposal presented in this paper generalizes some of our previous research [17], which established an algebraic formalization of the Objectcharts[3] notation [6], and presented the formal verification of the orthogonality of an Objectchart. Nevertheless, that work did not include such aspects as the verification of the orthogonality in diagrams with complex transitions which are essential in the specification of concurrent systems. This feature is now fully considered.

The effort of many research groups is now directed from classical graphical notations towards the UML notation. In this sense, one of the most active groups is the precise UML group [28] (pUML). This group is made up of international researchers and practitioners who are interested in providing a precise and well-defined semantics for UML, by using model-oriented notations, such as Z or VDM. The precise UML group has formalized a number of UML diagrams such as the Class Diagram [9], including some extensions for it [20], as well as reasoning on these diagrams [8]. The approach followed is the same as ours according to the metamodel formalization strategy: they provide

---

[3] Another OO extension of the David Harel's Statecharts notation.

schemas to formalize the UML modeling elements in such a way that it is not necessary to change those specifications to cope with the particular domain models. However, in our opinion, the UML metamodel evolution is not readily supported by the Z formal language used. On the contrary, the algebraic approach used in this paper can upgrade in a natural way to Maude specifications that include specific mechanisms to deal with extensibility and the metamodel evolution, e.g., reflection, possibility of dealing with theories and specifications as if they were also terms [31].

6: Conclusions and further work

The algebraic specification presented in this paper shows how the syntax and the static semantics of the UML Statechart Diagrams, as a part of the UML metamodel, can be modeled, animated and formally verified.

By using an algebraic specification language – Maude - the above-mentioned mathematical models have been made executable. Taking advantage of the underlying mathematics and using the executability provided by Maude, as an automatic deductive system, a variety of properties related to a given UML Statechart diagram specification can be stated and proven by software developers. In order to illustrate the procedure to formalize and verify these kinds of properties, a desirable characteristic of the UML Statechart Diagrams, namely orthogonality, has been modeled in the same framework. Thus, any particular UML Statechart diagram can be tested to ensure whether it fulfils the orthogonality property. This procedure for the UML Statechart Diagrams can be extended in the same way to any other property identified as interesting for users or developers. Having obtained this mathematical model for the UML Statechart Diagrams, the next task to be undertaken is to complete it by formalizing its dynamic semantics within the same algebraic theory. Despite this shortcoming, the current formal model enables us to rigorously verify the syntax and the static semantics of the domain model. We have put this formal model into practice by integrating it into an OO prototyping and verification process model [23], aimed at the OO Analysis and Design model verification, and Eiffel code automatic generation.

We are continuing to search for outstanding semantics properties and subsequently to formalize them in the same framework, in order to be rigorously tested. Checking the fulfillment of the multiplicity property of an *association-end*, generating automatically derived associations or generating a minimal intended model from an analyst-defined model may be examples

of this. At the present, the authors have started a number of contacts with Industry to apply these results to ongoing projects that involve the UML Statechart Diagrams specifications among their deliverables. We are of the opinion that this realm should be widely exploited.

This work is currently being extended to achieve a whole formalization of the rest of the UML diagrams; preliminary versions of the UML Class Diagram (including its dynamic semantics) [12] and of the constraint language OCL are already available, as well as a proposal [10][31] to formalize the evolution of the UML metamodel based upon the reflection, a property of rewriting logic which has been efficiently implemented in Maude.

Acknowledgments

References

1. E. Astesiano and G. Reggio, "Algebraic specification of concurrency", 8ᵗʰ Workshop on Specification of Abstract Data Types joint with the 3ʳᵈ COMPASS Workshop, August 1991, pp. 1-39.

2. E. Battiston, F. D. C. Battiston and G. Mauri, "Modular Algebraic Nets to Specify Concurrent Systems", IEEE Transactions on Software Engineering, vol. 22, no. 10, October 1996.

3. M. Bettaz. "An association of algebraic term nets and abstract data types for specifying real communication protocols" 7th WADT, Lecture Notes in Computer Science, no. 534, Springer-Verlag, Berlin, 1991, p. 11-30.

4. R. H. Bourdeau and B. H. C. Cheng, "A formal Semantics for Object Model Diagrams", IEEE Transactions on Software Engineering, Vol. 21, no. 10, October 1995, pp. 799-821.

5. Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, Veronika Thurner, "Towards a Formalization of the Unified Modeling Language", Proc. 11ᵗʰ European Conference Object-Oriented Programming (ECOOP'97), Springer, Berlin, Lecture Notes in Computer Science, no. 1241, 1997, pp. 344-366.

6. D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to Use Objectcharts in Object-Oriented Desing", IEEE Transactions on Software Engineering, vol 18, no. 1, January 1992, pp. 9-18.

7. D. Craigen, S. Gerhart and T. Ralston, "Formal Methods Reality Check: Industrial Usage", IEEE Transaction on Software Engineering, vol. 21, no. 2, February 1995.

8. A. S. Evans, "Reasoning with UML class diagrams", WIFT'98, Boca-Raton, IEEE Press, 1998.

9. A. S. Evans, A. N. Clark "Foundations of the Unified Modeling Language", Proc. of the 2nd Northern Formal Methods Workshop, Springer-Verlag, 1998.

10. J.L. Fernández, A. Toval, "Interested in Formalizing the UML Evolution Semantics? Be "Reflective", Workshop "Rigorous Modeling and Analysis of the UML Challenges and Limitations", OOPSLA'99, Denver, Colorado, 2ⁿᵈ November, 1999.

11. J. L. Fernández and A. Toval, "A Formal Syntax and Static Semantics of UML Statecharts" Technical Report LSI 3-99, Dept. Informatics and Systems, University of Murcia, Spain, 1999.

12. J. L. Fernández and A. Toval, "Fomally Modeling and Executing the UML Class Diagram" Proc. of the V Workshop MENHIR (Models, Environments and Tools for Requirements Engineering) (M.J. Rodríguez, P. Paderewski eds.) March 30-31, 2000, University of Granada, Spain.

13. R.B. France, J.-M. Bruel, and M. M. Larrond-Petrie, "An Integrated Object-Oriented and Formal Modeling Environment", Journal of Object-Oriented Programming,. November/December 1997, pp. 25-34.

14. M. Gogolla, F. P. Presicce, "State Diagrams in UML: A Formal Semantics using Graph Transformations", Workshop on Precise Semantics for Software Modeling Techniques, ICSE 1998.

15. J. A. Goguen and G. Malcom, "Algebraic Semantics of Imperative Programs" MIT Press 1996.

16. J. A. Goguen and G. Malcom, "Software Engineering with OBJ. Algebraic Specification in Action", Kluwer Academic Publishers 2000.

17. B. A. Grima and A. Toval, "An algebraic formalization of the Objectcharts notation", Proc. of GULP-PRODE 1994, the 1994 Joint Conference on Declarative Programming, Publishing Department of Technical University of Valencia, Spain.

18. D. Harel, "Statecharts: a visual formalism for complex systems", Science of Computer Programming, vol. 8, 1987, pp. 231-275, North-Holland.

19. D. Harel, "On Visual Formalism", Comm. of the ACM, vol. 31 (5) 1.988.

20. S. Kent, "Constraint diagrams: visualising invariants in {OO} models", OOPSLA'97, ACM Press, 1997.

21. J. Meseguer, "A logical theory of concurrent objects", Proc. of ECOOP-OOPSLA'90 Conf. on Object-Oriented Programming, Ottawa, Canada, ACM, October 1990, pp.101-115.

22. J. Meseguer, "Conditional rewriting logic as a unified model of concurrency". Theoretical Computer Science, 96(1):73-155, 1992.

23. B. Moros, J. Nicolás, J. G. Molina A. Toval, "Combining Formal Specifications with Design by Contract", Journal of Object-Oriented Programming. Vol. 12, no. 9, February 2000.

24. Object Management Group (OMG), "UML Notation guide version 1.3", June 1999, http://www.rational.com/uml.

25. Object Management Group (OMG), "UML semantics version 1.3", June 1999, http://www.rational.com/uml.

26. G. Övergaard, "A Formal Approach to Relationships in the Unified Modeling Language", Workshop on Precise Semantics for Software Modeling Techniques, ICSE 1998.

27. C. Pons, R. Giandini and G. Baum, "A tool for verifying formally graphical specifications of objects" (In Spanish),

Proc. of the II Conference on Software Engineering, San Sebastián (Spain), pp. 193-206.

28. The precise UML group. http://www.cs.york.ac.uk/puml/.

29. G. Reggio and R. J. Wieringa, "Thirty one Problems in the Semantics of UML 1.3 Dynamics", Workshop "Rigorous Modeling and Analysis of the UML Challenges and Limitations", OOPSLA'99, Denver, Colorado, 2nd November, 1999.

30. A. Toval, I. Ramos, O. Pastor, "Prototyping Object Oriented Specifications in an Algebraic Environment", in Database and Expert Systems Applications, Lecture Notes in Computer Science, no. 856 (D. Karagianis, ed.), Springer-Verlag 1994.

31. A. Toval and J.L. Fernández "Formally Modeling UML and its Evolution: a Holistic Approach", Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), Stanford, California, USA, Proc. by Kluwer Academic Publishers.September 6-8, 2000.

32. E. Y. Wang, H. A. Richter, and B. H. C. Cheng, "Formalizing and Integrating the Dynamic Model within OMT", Proc. of IEEE International Conference on Software Engineering, May 1997.

33. R. J. Wieringa and G. Saake, "A Formal Analysis of the Shlaer-Mellor Method: Towards a toolkit of Formal and Informal Requirements Specification Techniques", Requirements Engineering Journal 1996, vol. 1, pp. 106-131.