An Extension of Logical Formalisms to Simulate System Behaviours

Alain Cougoulic

France Télécom
Branche Développement
Centre National d'Études des Télécommunications/DES/ERA
28, Chemin du Vieux Chêne
F-38243 MEYLAN CEDEX, FRANCE
Tel: 33 4 76 76 44 08
Fax: 33 4 76 76 43 39
Email: Alain.Cougoulic@cnet.francetelecom.fr

Abstract

Using formal methods to specify systems becomes more and more a necessity to get round the increasing system complexity. In this paper, we discuss the use of formal methods to specify system behaviours. Behaviours represent the dynamic part of a system. System behaviours are often specified with logical formalisms.

Our interest in component-based software engineering is focused on the validation of system specifications by simulation. The simulation of system dynamics as described by behaviours consists in determinating and executing the *animate* part of the system behaviour specifications. We show how the addition of *explicit determinism* in the behaviour specifications (in logical formalism) allows one to increase the *animated* part of the system behaviour specifications.

Keywords: System specifications, behaviours, simulation, logical formalisms, determinism

Workshop Goals: Discussion about system behaviour specifications; advance of refinement of specifications; interest of simulation in component-based software engineering

1 Background

Using formal methods to specify systems becomes more and more a necessity to get round the increasing system complexity. In this paper, we present the use of formal methods throughout system specifications. More precisely, our work is based on TMN (Telecommunication Management Network), so the subject matter being the use of formal methods to specify distributed systems.

Distributed systems are often represented as a set of objects distributed on networks. An object is composed of some data and offers some services. An object specification is an abstraction of object data and services which is independent of its implementation. Recently, the distributed systems development is based on the client/server paradigm. The client/server paradigm represents communication between two processes, the client and the server, the client requiring some services given by the server. In this paradigm, distributed systems are specified in terms of interfaces between objects. Some current specification languages (CORBA/IDL [COR93], DCE/IDL [KTW93], ODL [ODL96], etc) base their system representations on this paradigm. The IDL (Interface Definition Language) generic name is associated to these languages.

However, specifying object interfaces is not sufficient because IDL languages do not specify the system behaviour, or even the object behaviour. Object behaviours are related to object states. An object state is the set of data values and offered services that the object can deliver at a certain time. In fact, an object life is a sequence of states. In object state terms, an object behaviour can represent :

- an object invariant,
- an object state change,
- output values of object services.

An object invariant is a set of properties that are respected by all object states. For example, the number of subscribers that are referenced by a telephone dispatcher cannot exceed the dispatcher capacity.

An object state change is a set of properties that must be respected when an object switches state. A state change can be a consequence of :

- an object service out going call,
- an object service in coming call,
- a climatic event, or other non-formal events,
- etc

For example, the addition of a subscriber on a telephone dispatcher is an object state change behaviour.

An object services can have parameters, and particularly output parameters. During a service call, their output parameters take new values, so an object behaviour can specify these output values.

In the telecommunications world, object behaviours are defined by some specification languages (GDMO [GDM92], SNMP/MIB [RM90], etc), however, in these languages, behaviours are described informally. Our work is to simulate system specifications, so formalizing objects behaviour is of interest to us.

Object behaviour specification can be formulated using logical formalisms. An object state can be described by a formula if the formula is made true when the logical constants of the formula are mapped in the state computer variables, and we say that a state satisfies a formula if that formula describes it [Mor90]. In this context, an object invariant is the common logical assertion that the different states of the object satisfy, and an object service behaviour is a pair of pre and post-conditions (logical assertions). A pre-condition describes the object state before a service call, a post-condition describes the object state and the output values after a service call. An interpretation of a pair of pre/post-condition is : if the previous object state satisfies the pre-condition then the next object state satisfies the post-condition, otherwise we do not know what will happen.

To illustrate the use of logic to formalize object behaviours, we give the example of a telephone dispatcher object invariant and the pre and post-conditions description [1] of its state change when the addition of a subscriber is realized :

- $n \leq l$
- $[n < l \ , \ n' = n + 1]$

where n represents the number of subscribers that are constantly referenced by the telephone dispatcher, l is the capacity of the dispatcher, n and l are computer variables, but in logic, n and l are logical constants. In pre and post-condition description, an occurrence of n is *decorated*. This occurrence represents the value of n after the change of object state[2], the undecorated occurrences of n represent the value of n before the change of object state[3]. Finally, only decorated variables are changed by a service call.

The use of logical assertions to specify object behaviours is compatible with existing distributed system specification languages. Many specification languages are defined on the object paradigm. Object oriented descriptions permit reuse, transparency, etc. For example,the inheritance concept maps reuse in object oriented descriptions. The above mentioned languages (CORBA/IDL, GDMO, ODL, etc) all respect inheritance of *specialization* [Boo92], i.e. a daughter interface is completely compatible with its mother interfaces. Figure 1 represents inheritance of object behaviour specified with logical assertions (in this figure, the $\Rightarrow$ symbol represents logical implication).

When a daughter interface inherits from a mother interface, the daughter interface behaviour is a composition of the mother interface behaviour, so the composition mechanism of logical assertions must respect the inheritance of specialization. For this, the composition mechanism proposed by EIFFEL [Mey92] respects the inheritance of specialization, and is reviewed in these expressions :

- $Inv2 = Inv1 \wedge AddInv$, by definition of conjunction, $Inv2$ is greater or equal to $Inv1$,
- $Pre2 = Pre1 \vee AddPre$, by definition of disjunction, $Pre2$ is smaller great or equal to $Pre1$,

[1] a pair of pre/post-conditions is represented by $[pre \ , \ post]$.

[2] we use the terminology of the Z language [Spi92].

[3] only post-conditions reference some decorated occurrence.

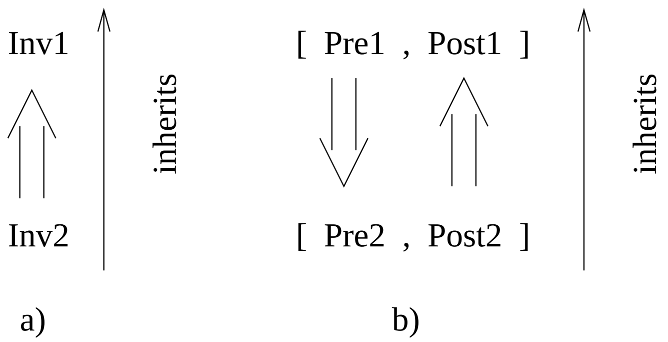

Figure 1: Inheritance of specialization a) of object invariants b) of pre/post-conditions

- $Post2 = Post1 \wedge AddPost$, by definition of conjunction, $Post2$ is greater or equal to $Post1$.

2 Position

The inheritance of specialization is a particular term of the object world. In the programming world, we talk about refinement of specifications [Mor90]. Refinement of specifications is used to imply proof of programs. In the simulation context, where we are interested to validate specifications, we talk about prototyping applications rather than refinement of specifications. The problem described in this paper is the efficient use of a logical formalism to specify objects behaviour for defining *prototypes* of applications through the simulation of specifications. Our interest is not restricted to the use of a particular logical formalism, so the logical formalism that we describe in subsection 2.1 is a general view of traditional logical formalisms. In subsection 2.2, we present the problem, and our solution is defined in subsection 2.3. Subsection 2.4 presents the interesting points of our solution in component-based software engineering.

2.1 A simple logical formalism

The vocabulary of our simple logical formalism is composed of :

- $\wedge$, $\vee$, $\neg$ connectors and $\forall$, $\exists$ quantification symbols,
- a set of logical variables and a set of logical constants[4],
- a set of predicates (equality, etc).

The values domain of logical variables and constants is composed of integers, strings, sets, lists, tables, etc. Before defining logical formulae, we define terms that compose a logical formula :

- a logical variable is a term (if this variable is referenced under a quantification symbol),
- if f is a logical constant in n arguments and $t_1, ..., t_n$ some terms then $f(t_1, ..., t_n)$ is a term.

[4]in the following, we name also these kinds of logical constants *specification variables*.

Definition of logical formula :

- if P is a predicate in n arguments and $t_1, ..., t_n$ some terms then $P(t_1, ..., t_n)$ is a (atomic) formula,
- if A is a formula then $\neg A$ is a formula,
- if A is a formula, t a term and x a logical variable then $(\forall x \in t \mid A)$ and $(\exists x \in t \mid A)$ are formulae,
- if A and B are formulae then $A \wedge B$ and $A \vee B$ are formulae.

We introduce another notation often used in logic and named the *conditional operator*. The logical definition of it is :

$$\text{IF } c \text{ THEN } a \text{ ELSE } b \quad \rightarrow \quad (c \wedge a) \vee (\neg c \wedge b) \tag{1}$$

2.2 Problem

The simulation of specifications permits one to finalize specifications (i.e. to bring them to there final version) or to validate system implementations with regard to its specifications. We are interested in finalizing specifications. In the case of a service call (when an object state switches), the post-condition related to it references object specification variable values and how they change during the call execution. Only post-conditions define these changes because if a pre-condition defines many value change, then this pre-condition is a post-condition of an other service call. An example is the couple of pre/post-conditions $[n < l, n' = n + 1]$ defined in section 1, the post-condition $n' = n + 1$ specifies the change of n value during the addition of a subscriber to a telephone dispatcher.

When we simulate specifications, we want to *animate* the object value changes on the simulated object states. We classify these value changes in two categories :

- a specification variable takes only one new value,
- a specification variable must choose between some new values.

For example, these two categories of value changes are presented in this post-condition :

$$x' = 1 \wedge y' < 6 \tag{2}$$

The $x' = 1$ sub formula defines only one possible new value for x, the value 1. This sub formula is *deterministic*. The $y' < 6$ sub formula defines a set of possible new values for y, this set is {0,1,2,3,4,5}. This sub formula is *non-deterministic*. The *deterministic* notion is associated with the *substitution* notion. A substitution is a set of pairs of variables and values. A substitution e is defined on a formula f if the set of variables in the substitution e is the set of variables in the formula f. The definition 1 give a possible definition of the determinism of a formula.

Definition 1 *Let f be a formula, f is deterministic if :*

$$\text{For all } (x_1, v_1) \in e_1^f \text{ and } (x_2, v_2) \in e_2^f \text{ with } x_1 = x_2 \text{ then } v_1 = v_2 \tag{3}$$

where e_i^f is a substitution and e_i^f satisfies f[5].

[5] a substitution satisfies a formula if, after substitution, the formula is evaluated to true.

From a practical point of view, the treatment of the deterministic part of a formula is more efficient. If a post-condition is deterministic then the object related to this post-condition can only have one state after execution of the service related to this post-condition. Otherwise the object can be in many different states, and we must choose between these states, or study all possibilities. In fact, the main problem discussed in this paper concerns the distinction between the determinism and the non-determinism of the formula.

To separate the determinism from the non-determinism of a formula seems to be trivial in formula 2. We can imagine how to define an algorithm that computes the separation between the determinism and the non-determinism of a formula. However, our simple logical formalism defined in subsection 2.1 does not restrict the set of predicates, in particular, we use some undecidable predicates, in this case, all algorithms are also undecidable. An element of solution is the notion of *explicit determinism* which we describe in subsection 2.3.

2.3 Explicit determinism

The definition of explicit determinism follows its name. Determinism is marked in the syntax of the formula. Advantages of explicit determinism are :

- trivial decidability,
- easy use.

To mark explicit determinism, we introduce a new special predicate represented by the $:=$ operator. The $:=$ predicate is used in GDMO$^+$[GDM97]. The set of atomic logical formulae made on this predicate is a subset of the set of atomic logical formulae made on equality predicate.

The $:=$ predicate does not specify all determinism of a logical formula (for example $0 < y \wedge y < 2$ is deterministic in the set of natural integers). This predicate can be introduced in every logical formalism but some constraints are imposed. For example, the formula :

$$x' + y' := 4 \tag{4}$$

is not allowed because it is not deterministic, and is in contradiction with the $:=$ predicate intention to mark determinism.

Constraints on the $:=$ predicate can be separated into two categories :

- constraints on atomic formulae defined with the $:=$ predicate ;
- constraints on complex formulae[6] containing an atomic formula defined with the $:=$ predicate.

Constraints on atomic formulae The $:=$ predicate is represented by a symbol often used to define assignment in many programmation languages. In the case of simulation, we decide to consider this predicate as an assignment. In the first place, we define a direction of evaluation of an atomic formula based on the $:=$ predicate. We *assign* the right operand

[6]a complex formula is composed of some atomic formulae that are linked by some connectors or some quantification symbols or some conditional operators.

to the left operand. In the second place, only decorated variables can be modified, so the left operand is composed of exactly one decorated variable.

Finally, let's take the example of the formula :

$$x' := y' + 2 \tag{5}$$

This formula is not deterministic because we do not known the value of y'. Furthermore, not any decorated variables must appear in the right operand.

We summarize the syntax of an atomic formula based on the := predicate by these generic formulae :

- $x' := ...$
- $f'(p_1, p_2, ...) := ...$

where not any decorated variables appear in any p_i.

Constraints on complex formulae The $\vee$ and $\neg$ connectors are by definition non-deterministic, so an atomic formula based on the := predicate cannot be an operand of these connectors. On the contrary, the $\wedge$ connector is by definition deterministic, so an atomic formula based on the := predicate can be an operand of this connector. Some examples :

- $x' := 1 \vee x' := 2$ is invalid.
- $\neg(x' := 1)$ is invalid.
- $x' := 1 \wedge x' := 2$ is valid.

The last formula is declared valid by the determinism evaluation. Logical evaluation of this formula is false.

An universally quantified formula can be deterministic, so an atomic formula based on the := predicate can be in the scope of an universally quantified formula. To achieve this, the iterative set related to the universally quantified formula must be known, i.e. no decorated variable occurrence is referenced in this iterative set. For example, a valid universally quantified formula and containing an atomic formula based on the := predicate is :

$$\forall i \in \{1, 2, 3, 4\} \mid f'(i) := f(i) + 1 \tag{6}$$

where f is a table.

In contrast to an universally quantified formula, an existentially quantified formula is not deterministic. Such formula can be evaluate to true for one or many values defined in its iterative set. A possible choice between these values enforces no determinism of an existentially quantified formula. The formula :

$$\exists i \in \{2, 3, 4, 5\} \mid (i \text{ modulo } 2) = 0 \ \wedge \ f'(i) := 5 \tag{7}$$

is invalid because two iterative values (2 and 4) evaluate the formula to true, furthermore, we do not know which of $f(2)$ value or $f(4)$ value must be modified.

An atomic formula based on the := predicate can be referenced in THEN and ELSE branches of a conditional operator if the condition of the formula based on the conditional operator is known, i.e. no decorated variable occurrence is referenced in this condition. For instance, this formula :

$$\text{IF } x = y \text{ THEN } x' := 1 \text{ ELSE } x' := 2 \tag{8}$$

is valid.

Our approach is practical, so the constraints that we define in this sub section are very restrictive. This restrictive choice permits a syntactic solution to determine whether a formula is deterministic, i.e. we do not use semantic properties to determine if a formula is deterministic but merely a syntactic algorithm which determines if a formula is deterministic.

2.4 Interests in component-based software engineering

In the telecommunications world, the concepts or paradigms used to specify systems are not finalized, i.e. this world is in *perpetual motion.* The solution defined in this paper to specify objects behaviour can be applied to many specification languages that specify objects behaviour with a logical formalism.

Our solution permits to *animate* specifications with only the addition of a new predicate (:=) in any logical formalism. Animating specifications is an advantage to prototype applications (see subsection 2.2) and in automatic code generation (a part of the system code).

With explicit determinism, the automatic code generation is easier because it can be mapped in programming language that implements system. For examples, the := predicate can be considered as an assignment operation, and an universally quantified formula as an iteration operation.

However, in implementation terms, the expressiveness of the explicit determinism defined in this paper is sufficient. The constraints defined in subsection 2.2 are very strong. An example is this formula :

$$x' := 1 \wedge y' := 2 \wedge z' := x' + y' \tag{9}$$

This formula is not valid because $z' := x' + y'$ is not valid, but we can consider as valid with the use of a simple ordering algorithm.

A part of our work is to reduce the constraints of our solution to accept the previous kind of post-conditions.

3 Related Work

In the distributed systems world, many works formalizing the objects behaviour exist. These works can be separated into two categories :

- the FDTs (Formal Description Techniques) ;
- the extensions of existing distributed system specification languages.

An FDT is an existing language which is being used in other domains. Specifying object behaviours with a FDT language comes to impose the FDT language properties such as the typing of data on the application specifications ([Kel95], [Fes94]). Some FDT examples are VDM, LOTOS, Z, etc [ISO94].

In some distributed system specification languages, the object behaviours are defined informally (cf. section 1). In these languages, the object behaviours are explicitly identified in specifications. Some works replace these informal behaviours by formal descriptions. In contrast to FDTs, no constraints on typing of data are defined, since the extended specification languages do not constraint the languages extended. Some extended languages are GDMO$^+$ [GDM97], GDMO/CRS [Fes94], Q-GDMO-GRM [TIN95], etc.

None of the above mentionned solutions define animated specifications such as the use of explicit determinism. Furthermore, our work is to apply explicit determinism on Q-GDMO-GRM. Q-GDMO-GRM allows to the use of any logical formalism to specify object behaviours. Our practical work is to simulate Q-GDMO-GRM specifications extended with explicit determinism.

References

[Boo92] G. Booch. *Conception orient e objets et applications.* Addison-Wesley France, 1992.

[COR93] Object Management Group. *The Common Object Request Broker : Architecture and Specification, Rev. 1.2.*, 1993.

[Fes94] O. Festor. *Formalisation du comportement des objets g r s dans le cadre du mod le OSI.* PhD thesis, Universit Henri Poincar - Nancy I, 1994.

[GDM92] International Telecommunication Union. *Structure of Management Information: Guidelines for the Definition of Managed Objects, X.722*, 1992.

[GDM97] International Telecommunication Union. *Structure of Management Information: Guidelines for the Definition of Managed Objects, Draft Amendment 4, X.722*, 1997.

[ISO94] ISO/IEC JTC 1/SC 21/WG 4 N8088. *Working Draft on the Use of FDTs for the Specification of the Behaviour of Managed Objects*, 1994.

[Kel95] J. Keller. An extension of gdmo for formalizing managed objects behaviour. In *8th IFIP TC6 International Conference on Formal Description Techniques (FORTE'95), October 17-20, 1995, Montreal, Canada.*, 1995.

[KTW93] N. Kincl, D. Thompson, and R. Webber. *Adapting DME to Manage DCE-based Services, RFC 22.0.* The Open Group, 1993.

[Mey92] B. Meyer. *EIFFEL, The Language.* Object-Oriented. Prentice Hall, 1992.

[Mor90] Carroll Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[ODL96] Telecommunications Information Networking Architecture Consortium. *TINA Object Definition Language MANUAL, Version: 2.3*, 1996.

[RM90] M.T. Rose and K. MacCloghrie. *Structure and Identification of Management Information for TCP/IP-based Internets, RFC 1155.* Internet Engeneering Task Force, 1990.

[Spi92] J.M. Spivey. *The Z Notation.* Prentice Hall, 1992.

[TIN95] Telecommunications Information Networking Architecture Consortium. *Information Modelling Concepts, Version: 2.0*, 1995.

4 Biography

Alain Cougoulic receives a DEA degree in computer science at the University of Rennes I. He prepares a Ph.D from the Institut National Polytechnique de Grenoble. He researches in LSR (Logiciels Systèmes et Réseaux) laboratory of IMAG (Institut de Mathématiques Appliquées de Grenoble) associated with CNET (Centre National d'Études des Télécommunications), France Télécom. His Ph.D subject is the simulation of managed objects behaviour.