CS 524

Concurrent Objects in Rewriting Logic: Farther Reading Lecture 13

J. Meseguer

1. Modeling classes, subclasses and class Inheritance

Arrangement of objects into classes and subclasses is a very natural structuring mechanism, since it reflects the natural structure of things. For example, zoological or botanical taxonomies classify animals or plants in exactly this way, respecting natural barriers — for example in genetics, so that reproduction between animals in different species is usually impossible.

In software, classes and class inheritance is a powerful reusability mechanism: one can easily specialize an already defined object class into a useful subclass, which inherits the attributes and "methods" [in a concurrent setting, the "methods" are accounted for by messages] of the superclass.

The key question is:

> How can a formal semantics of classes and class inheritance for concurrent objects be given in rewriting logic?

This question is non-trivial for at least two reasons:

1. The notion of class inheritance is fuzzy, pre-theoretic,

and "polymorphic": it is understood in quite different ways in various programming languages. For example:

1. It is, essentially a notion of taxonomy in the Simula language, the first object-oriented language ever proposed, but
2. It becomes instead a notion of code extension and code modification in the Smalltalk language, which had also an important influence.

The issue of providing a formal semantics in rewriting logic to classes and class inheritance was addressed in the paper:

J. Meseguer, "A logical theory theory of concurrent objects and its realization in the Maude language," in G. Agha, P. Wegner and A. Yonezawa, Research Directions in Concurrent Object-Oriented Programming, 314-390, 1993, MIT Press.

In summary, the answer is based on the following basic insights:

1. The concept of inheritance, as it appears in practice, is overloaded and over-extended, giving rise to various confusions and to conflicting requirements.

2. Much clarity can be gained by sharply distinguishing two different notions of inheritance:

2.1 Class Inheritance, takes seriously the notion of taxonomy: it is monotonic, in the sense that all properties and functionality of a superclass is actually inherited without change or modification by its subclasses.

2.2 Module Inheritance is a different and orthogonal notion. It does justice to the practical need for code reuse*, code modification, and code adaptation. Rather than being viewed as an operation on classes, is instead view as an operation/transformation between software modules.

Furthermore, these notions are both clearly distinguished from the notion of parameterization, in the sense of parameterized modules [parametric polymorphism], since, unfortunately, all these notions can be confused together.

(*) Of course, class inheritance is also a powerful code reuse mechanism. The problems come when both notions are conflated.

3. The notion of class inheritance is giving a precise semantics in rewriting logic taking advantage of the fact that its underlying equational logic is order-sorted, so that:

3.1 a subclass $C' < C$ is precisely modeled as a subsort

3.2 all the rewrite rules of the original superclass C are inherited without change by the subclass C', but now:

3.3 { (a) object attributes
(b) messages, and
(c) rules }

can be added to define the objects of the subclass C'

3.3 The "magic" that ~~this~~ makes inheritance of rules possible is the fact that the operator

$$\_,\_ : \text{Attributes Attributes} \longrightarrow \text{Attributes}$$

is associative-commutative with identity. This means that a rule, say, like

$$\text{credit}(A, M)\ \langle A : \text{Accnt} \mid \text{bal} : \mathbf{N} \rangle \longrightarrow \langle A : \text{Accnt} \mid \text{bal} : M + N \rangle$$

of a superclass Accnt of bank accounts has really the form:

credit(A,M) $\langle$ A : Accnt | bal : N, ATTS $\rangle$ $\rightarrow$ $\langle$ A : Accnt | bal : M+N, ATTS $\rangle$

where ATTS is a variable of sort Attributes which stands for "any other attributes" that the object might have.

Of course, the objects of class Accnt may only have the bal(ance) attribute, so ATTS for them will always be the empty set of attributes. But this makes the above rule extensible to any future subclass, such as a CheckingAccnt subclass with, say, a checking-history attribute added.

4. The notion of module inheritance is different: it takes care of the need, that sometimes arises, for method specialization, i.e., that the behavior of a method/message on a subclass is somewhat different [changes the values of attributes, including new attributes, in a somewhat different way]. In rewriting logic and Maude this is viewed as a module transformation of the form:

$$(\Sigma, E \cup B, R_1 \cup R_2) \mapsto (\Sigma', E' \cup B', R_1 \cup R_2' \cup R_3)$$

where: 1. New sorts, corresponding to new classes are added

credit(A,M) $\langle$ A : Accnt | bal : N, ATTS $\rangle$ $\rightarrow$ $\langle$ A : Accnt | bal : M+N, ATTS $\rangle$

where ATTS is a variable of sort Attributes which stands for "any other attributes" that the object might have.

Of course, the objects of class Accnt may only have the bal(ance) attribute, so ATTS for them will always be the empty set of attributes. But this makes the above rule extensible to any future subclass, such as a CheckingAccnt subclass with, say, a checking-history attribute added.

4. The notion of ~~class~~ module inheritance is different: it takes care of the need, that sometimes arises, for method specialization, i.e., that the behavior of a method/message on a subclass is somewhat different [changes the values of attributes, including new attributes, in a somewhat different way]. In rewriting logic and Maude this is viewed as a module transformation of the form:

$$(\Sigma, E \cup B, R_1 \cup R_2) \mapsto (\Sigma', E' \cup B', R_1 \cup R_2' \cup R_3)$$

where: 1. New sorts, corresponding to a new classes are added

2. New rules R'_2 defining the behavior of messages previously defined by R_2 for the class C that is now specialized to a class C' are added,

3. New messages for C' can also be added, with semantics given by rules R_3, and

4. The rules $R_1 \cup R_2$ of the original module as well as its original classes are preserved.

2. Solving the "Inheritance Anomaly"

That this conceptual distinction between the two different mechanisms of: (a) class inheritance, and (b) module inheritance was not only useful by providing a clear mathematical semantics for a rather murky, confused, and pre-scientific notion, but was also useful in programming practice was demonstrated by solving an old chestnut in concurrent object-oriented programming, the so called inheritance anomaly. This anomaly was the constant, painful experience that:

a. code reuse for concurrent object systems, and
b. class inheritance

seemed to work [not work] at cross purposes: it seemed extremely difficult to have both feature simultaneously: when a class was extended, the code handling the messages of the superclass broke down when trying to handle the same messages for the subclass.

This anomaly was both explained and resolved by using the semantics of concurrent objects and the distinction between class inheritance and module inheritance in:

J. Meseguer, "Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming," Proc. ECOOP'93, Springer LNCS 707, 220-246, 1993.

The above theory of concurrent objects in rewriting logic has been implemented in the Full Maude extension of Maude. For all the details see:

Chapter 19: Object-Oriented Modules, in the Maude book: "All About Maude", Springer LNCS 4350, 2007.

From Concurrent Object System Design to Distributed Implementation

The Maude interpreter is the perfect place to design, prototype, analyze, and [using Maude's formal tools] verify a concurrent object system. However, within a single Maude interpreter concurrency is only simulated by considering, for example, all the interleaving computations of a concurrent system using Maude's search command. But,

> In which sense is Maude a declarative concurrent programming language?

In the precise sense that:

1. Concurrent object systems in Maude can be deployed across several machines, each containing some of the objects of the entire system, ~~with objs~~ and each running a different instance [or several instances] of the Maude interpreter so that:

   (a) message passing communication between two objects residing in the same machine and interpreter takes place by the usual rewriting process, but

(6) message passing between two objects residing in different interpreters and possibly in different machines takes place by the following extended rewriting process, based on rule-programmable Maude I/O TCP/IP sockets:

1. a message (m) sent by object o to object o' residing in a different interpreter is
   a. transformed into a string $\overline{m}$
   b. sent to an I/O socket object supporting communication between these two interpreters
   c. received by an I/O socket object supporting the same communication in the other interpreter
   d. converted back into its term representation m, and
   e. rewritten together with o' into a new state of o' and possibly new objects and messages in the other interpreter.

The key points are that:

A. 1. m going out of the first Maude interpreter into a socket, and
   2. m entering the second Maude interpreter configuration (configuration/state)

are both performed by so-called external rewriting (erewrite)

B. The entire process is defined by rewrite rules in Maude.

Note that this means that the configuration of objects and messages that on a single interpreter is just modeled as a multiset becomes a concurrent data structure where:

1. all objects and some messages are distributed across different local configurations in different Maude interpreters and machines, and
2. some messages m exist in transit across a socket as $\overline{m_0}\ \overline{m_1}\ \overline{m_2}$ where: $\overline{m_0}$ is the prefix string already received by the I/O socket in the target Maude interpreter, $\overline{m_2}$ is likewise the suffix pending in the I/O socket of the source interpreter, and $\overline{m_1}$ is the fragment in transit between both sockets.

A key result is that the deployment of a concurrent object system $\mathcal{R}$ as a distributed implementation is a correct by construction transformation $\mathcal{R} \mapsto \mathcal{D}(\mathcal{R})$ described in detail in:

S. Liu, A. Sandur, J. Meseguer and P. Ölveczky, "Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs," Proc. NASA Formal Methods Symposium, Springer LNCS 12229, 22-40, 2020.

In particular this means that all temporal logic properties verified for the design $\mathcal{R}$ are satisfied by its distributed deployment $\mathcal{D}(\mathcal{R})$.