Two-Level Modeling

Anthony Lauder, Stuart Kent
Computing Laboratory, University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, UK
Anthony@lauder.u-net.com, S.J.H.Kent@ukc.ac.uk

Abstract

Experience on a large banking application has highlighted expressive weaknesses in the standard (concrete) syntax of UML, resulting in models that are under-constrained. It transpires, however, that the abstract syntax meta-model underlying UML is inherently more expressive that the concrete syntax layered on top of it. By directly reaching into, exploiting and, where necessary extending the meta-model we are able to constrain fully our domain models. Furthermore, since different fragments of a given model require different levels of expressive power, we are able to utilize a blend of concrete and extended abstract syntaxes to achieve a compact yet rich form of modeling. Finally, enhancing the concrete syntax of the modeling language, allowing the association of new concrete graphical icons with our abstract syntax extensions, facilitates the expression of models in an even more compact, readable, and intuitive form.

1. Introduction

The origins of this paper lie in two years work (1996 and 1997) by one of the authors at a large international bank. There we were modeling (in UML) and implementing (in C++) a securities settlement system. It transpired that there were a number of important requirements that we simply could not express using standard UML syntax. This meant that the requirements for our system were effectively under-expressed, which considering the mission-criticality of the application to the bank was a serious matter.

A number of realizations led to a problem resolution. The first realization was that underlying the syntax of UML is a highly expressive meta-model – the full power of which is not fully reflected in the syntax of the language. The second realization was that it is possible to model directly from the UML meta-model without being constrained by the limits of the standard syntax, and this has the potential to enhance the richness of a model significantly. Unfortunately, expressing a model directly from the meta-model is tedious and time-consuming. This led to the third realization, which was that it is, in fact, both possible and desirable simultaneously to call upon the concrete syntax and exploit the meta-model directly within a single model. Using this strategy, most aspects of a given model can be modeled as normal via UML's concrete syntax, but those aspects for which standard UML syntax is insufficiently expressive can be expressed by appealing directly to the meta-model itself. The two-level modeling approach, we believe, leads to models which are rich in expressive power, relatively compact, and potentially amenable to CASE-tool interpretation.

Subsequent to the development of the settlement system, both authors of this paper collaboratively re-examined and reflected upon the issues outlined above. This joint effort has resulted in numerous important improvements to the original solution, and it is the results of this collaborative effort that are the main focus of this paper.

2. Securities settlement

2.1. Introduction

Imagine a banking system which settles security trades. Security trades are sale and purchase agreements between securities traders termed parties to the trade. Each trader terms the other party to the trade the *counter-party*. A trading agreement is essentially an agreement that one party will sell a given security to a counter-party for an agreed price on an agreed trade date. An agreed trade must eventually be settled, wherein securities and cash are exchanged, recorded and reported.

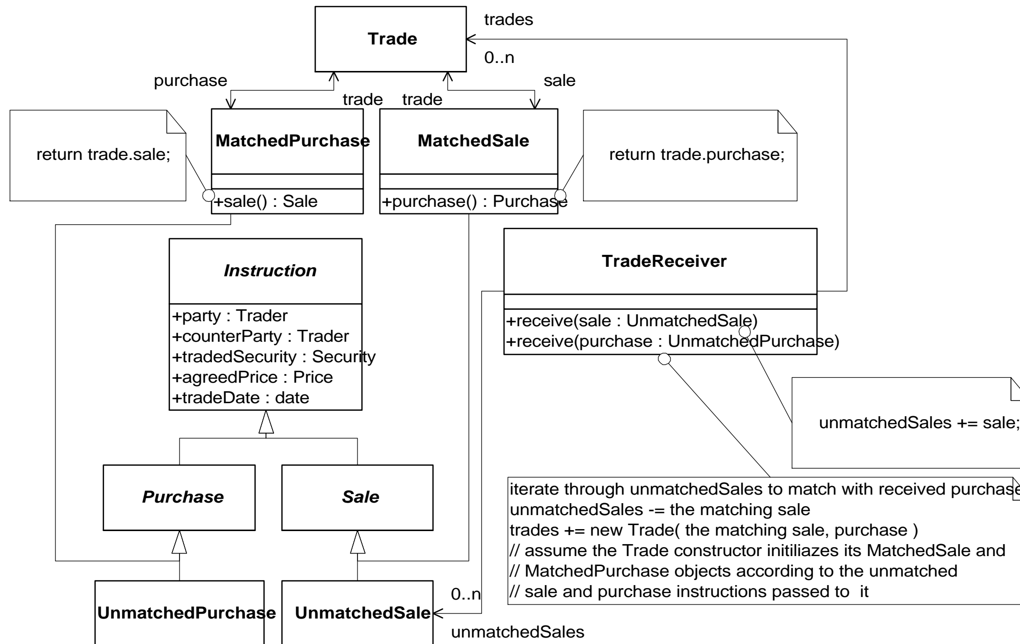

Figure 1. Trade Instruction Receipt

For a given trade, a settlement system receives two electronic instructions: a sale instruction from the selling party, and a purchase instruction from the purchasing party. Let us assume that the sale instruction is always received first, and that the counter-party's purchase instruction is awaited and will always arrive. Two instructions are said to match if their security, price, trade date and respective counter-parties are all in agreement. The first task of a settlement system, then, is to receive unmatched trade instructions and match them with their counter-party instructions. Matched instructions are bound together under a Trade object depicting the agreed trade common to them both. Figure 1 models the situation so far.

A Trade object goes through various settlement stages, which we will reduce to three: Processing, Recording, and Reporting. Processing reduces the buyer's cash, increases the seller's cash balance, and switches the traded securities from the seller to the buyer. Recording makes a permanent record of the processed trade. Reporting notifies the appropriate banks to transfer cash and certificates in accordance with the terms of the trade.

There are four possible states for a Trade object: JustMatched, Processed, Recorded, and Reported. We will use the State pattern [1] to track a Trade object through its stages of settlement. Each state has a method (named perform) which performs the next settlement stage and returns the next state (indicating completion of that stage). A Template Method [1] (named transition) in the base SettlementState class forms the public interface to these perform methods, and ensures that the old state is correctly deleted once a new state has been transitioned to. This is depicted in Figure 2.

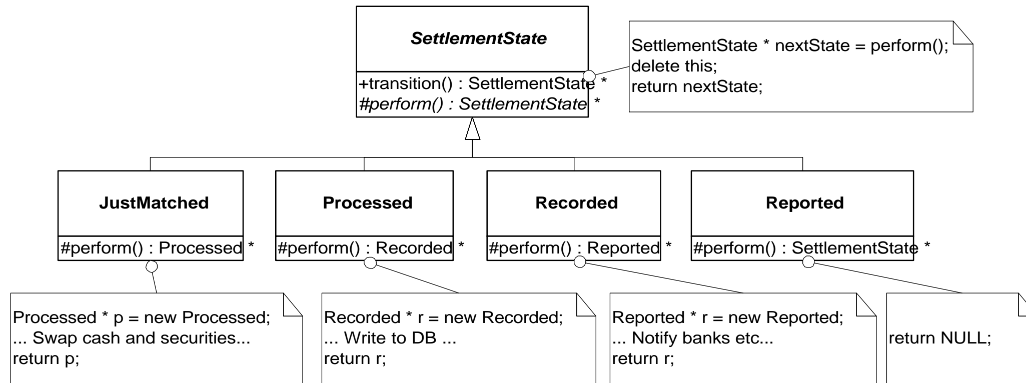

Figure 2. Settlement States

2.2. Trade class

Enhancing the Trade class with a reference to the current SettlementState (Figure 3), we see that new Trade objects start in the JustMatched state. Settlement of a Trade involves applying the exported transition method for each successive SettlementState until settlement is complete. This mechanism enables new SettlementState classes to be added to the Trade class and existing SettlementState classes moved around or removed with code changes localized to the SettlementState classes themselves.

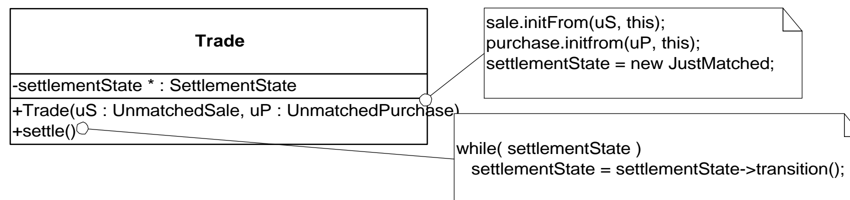

Figure 3. Trade Class with Settlement State

3. A serious omission

Unfortunately, there is a missing requirement. There is nothing in Figure 2 to prevent the addition of a class - derived from SettlementState - which, for example, returns a pointer to its current instance in perform. Consequently, when the transition method of Trade deletes an instance of such a class it would return that very same object (i.e. the one just deleted) as the new state. In cases like this, we are just heading for trouble. To prevent the addition of such badly behaved and dangerous classes, we need to add the explicit requirement that all implementations of perform behave according to the requirement that they either: (a) create and return a new SettlementState, or (b) return NULL.

UML distinguishes between a method (an implementation of behavior) and an operation (a specification of behavior). For each method there is assumed to be a corresponding operation (behavioral specification). So far, we have specified only methods, such as the implementations of perform attached (via little notes) to the various SettlementState classes in Figure 2. We could use the same approach (little notes) to express the omitted requirement as a specification for the perform operation on the base SettlementState class. Unfortunately, all this would really achieve is a comment that is readable by another human. What we really want is a way to express requirements that are enforcable by a CASE-tool. We see little point in adding requirement to a model if they cannot be enforced.

Note that many of the expressional omissions of UML have been compensated for by the addition of the textual constraint-expression sub-language OCL (the Object Constraint Language [3]). OCL, however, requires a certain degree of familiarity with concepts from the formal-methods community and may, therefore, prove somewhat inappropriate for modelers without such a background. In addition, the need to switch between visual and textual aspects of a model may be less appealing that an entirely visual model to some modelers. Instead of adopting a textual solution, as in OCL, this paper will propose an entirely graphical resolution to the expression of the omitted requirement.

4. Revealing the meta-model

4.1. The UML specification documents – an overview

The UML specification centers on two documents: The UML Notation [2] document focuses primarily upon the visual appearance of UML; describing the appearance of the graphical shapes (the *concrete syntax*) that a modeler can make use of when modeling. The UML Semantics document [4], on the other hand, focuses on the *meta-model* of UML. The meta-model is essentially a detailed description of the elements of the language without dictating a specific concrete syntax for them. The is achieved by "specifying the *abstract syntax* and *semantics* of UML." The abstract syntax (expressed in a subset of UML itself) defines UML's "constructs and their relationships", capturing the structural properties of the language elements and their interrelationships. In addition to abstract syntax, the Semantics document covers the semantics (i.e. the meaning) of the language elements.

The separation of abstract syntax from concrete syntax is both fortunate and important. We want to "lift the lid" on the UML specification, and reveal the abstract syntax within. Much of the weakness in the expressive power of UML derives not from the meta-model, but from UML's concrete syntax. The meta-model, it turns out, captures a more expressive language than is realized in the standard concrete syntax layered upon it. By passing by the concrete syntax, the richer underlying language contained within the meta-model is revealed. Revealing the abstract syntax enables us to not only model at the meta-level, but also to extend the meta-model itself. So that, if we are faced with a modeling task for which even the abstract syntax is insufficiently rich, we can simply extend the meta-model (using ordinary modeling techniques) thus enriching the modeling language it defines. Later in this paper, we will see these ideas used when we enhance UML to support enable expression of our missing requirement.

4.2. Abstract syntax of class

After several passes through the (unfortunately patchy) UML Semantics document, we have constructed a visual depiction of a fragment of the UML meta-model (see Figure 4).

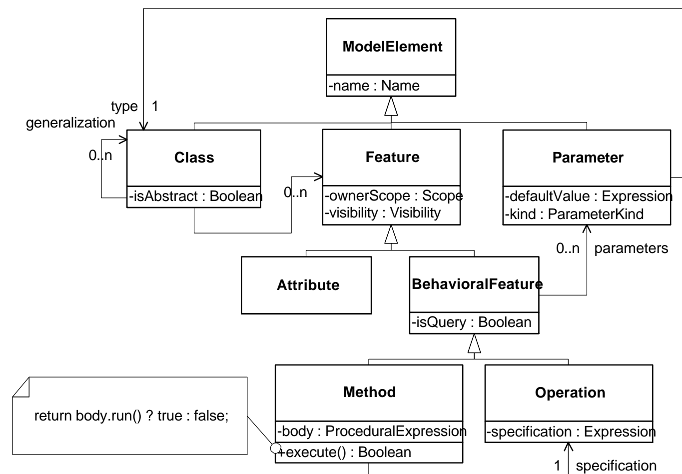

Figure 4. Abstract Syntax of Class

Figure 4 shows, somewhat simplified, the abstract syntax of a Class. The parts that are of particular interest to us in Figure 4 are the abstract syntax for Method and Operation. A Method is shown to have a body (a ProceduralExpression) which is executable. We have taken the liberty of supplying a method (execute), implied in the UML Semantics document, with which that body can be executed. Each Method has exactly one specification, which is an Operation providing a specification of type Expression.

4.3. Enhancing the meta-model to support enforced contracts

Unfortunately, the UML Semantics document lacks rigidity in its description of Operation specifications. All we really know, looking at Figure 4, is that Operation specifications are of type Expression (which turns out to be a very general type indeed). The UML Semantics document does state that "[t]he specification can be done in several different ways, e.g. with pre- and post-conditions, pseudo-code, or just plain text.", but provides no further details. This lack of clarity was unacceptable to us; we wanted a well-defined facility upon which to base our models. Thus we decided to enhance the meta-model of UML to support a more rigid form of Operation specification. Specifically, we derived a new type of Method (a CompliantMethod) and a new type of Operation (a ContractedOperation), as shown in Figure 5, which in combination permit the expression of methods governed by contracts which they are unable to disobey.

CompliantMethod captures the abstract syntax for a new type of method – one that must comply with its associated specification. Compliance is enforced by the execute method of CompliantMethod, which (if you look at the code) will only succeed in executing a defined method if that method satisfies its specification. It is this very property that we will take advantage later in the settlement system to ensure that the perform methods of classes derived from SettlementState are well behaved.

Each CompliantMethod, then, is associated with a specification with which it must comply. That specification is of type ContractedOperation, derived from the standard meta-level Operation class. Each ContractedOperation bears a contract (of type Contract – a new class we define). A Contract consists of a pre-condition and a post-condition. The pre-condition is a condition that must be true before the associated

CompliantMethod can start execution, and the post-condition is a condition which must be true when that execution ends. A contract specification may be a DisjunctiveContract, a ConjunctiveContract, or an AtomicContract. A DisjunctiveContract specifies two alternative Contracts, and an implementation is said to satisfy the DisjunctiveContract if it satisfies either, or both, of those alternatives. A ConjunctiveContract specifies two sub-contracts, both of which must be satisfied. An AtomicContract must be satisfied in full, and depicts a preCondition and a postCondition. In combination, the three forms of Contract enable the construction of sophisticated Contract hierarchies.

The preCondition and postCondition of an AtomicContract are both of type Condition. Condition exports a check method that returns a Boolean value based upon whether or not the associated Condition body is satisfied. A Condition body is expressed as a UML ObjectDiagram, and essentially depicts a set of objects that must exist for the Condition to be said to be satisfied.

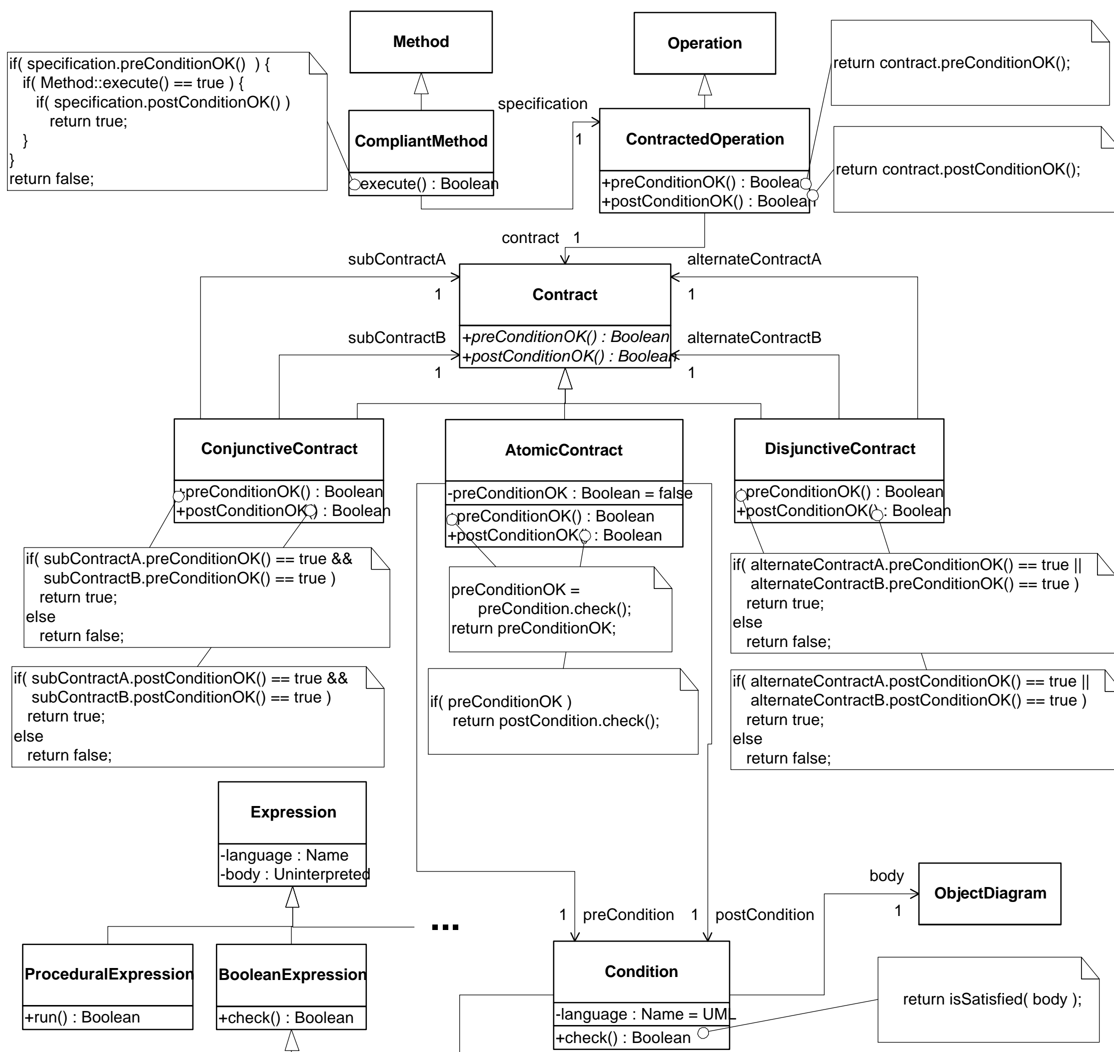

Figure 5. CompliantMethods and ContractedOperations

4.4. Redefining SettlementState using the abstract syntax

In Figure 6, we completely redefine our SettlementState class using the abstract syntax extensions of Figure 5.

Class
-name : Name = SettlementState
-isAbstract : Boolean = true

Parameter
-kind : Kind = return

Method
-name : String = transition
-ownerScope : scope = classifier
-visibility : Visibility = public
-isQuery : Boolean = false

Operation
-name : String = transition
-ownerScope : scope = classifier
-visibility : Visibility = public
-isQuery : Boolean = false

ContractedOperation
-name : String = perform
-ownerScope : scope = classifier
-visibility : Visibility = protected
-isQuery : Boolean = false

ProceduralExpression
-language : Name = C++

PseudoCodeExpression
-language : Name = pseudo code

DisjunctiveContract

SettlementState * nextState = perform();
delete this;
return nextState;

Get the next settlement state
delete self (current settlement state)
return the next settlement state

AtomicContract
Condition
SettlementState
NULL

type <reference to> parameters specification body contract alternateContractA alternateContractB preCondition postCondition <return value>

Figure 6. SettlementState Re-visited via AbstractSyntax

In Figure 6, we have depicted the transition method of SettlementState, which is not our prime concern here, via the normal abstract syntax for Method and Operation. Our real interest lies in the perform operation which is depicted as an instance of our new ContractedOperation class. Remember, in the SettlementState class we require only a

specification for perform; actual implementations of perform are deferred to derived classes. Hence, in the SettlementState class we see only a ContractedOperation for perform but no corresponding CompliantMethod. Perform's contract is a DisjunctiveContract, specifying two alternate Contracts (which in this case are AtomicContracts). The first alternate shows the existence of a SettlementState in its postCondition that does not exist in the preCondition. For a perform method to satisfy this preCondition/postCondition pair, then, it must have created a new SettlementState. In addition, that newly created SettlementState is shown to be referred to in the return value of perform. The second alternate AtomicContract simply allows perform to return a NULL. All implementations of perform must satisfy one of these two alternate contracts, otherwise those implementations will not be permitted to execute.

We can now add the actual settlement state classes (from JustMatched through to Reported) that are derived from the abstract SettlementState class. Since Figure 6 is already getting rather large and so we re-express a fragment of Figure 6 in Figure 7 and add those actual settlement state classes there. Note that only two settlement state classes are shown, since the approach would be basically identical for all other settlement states.

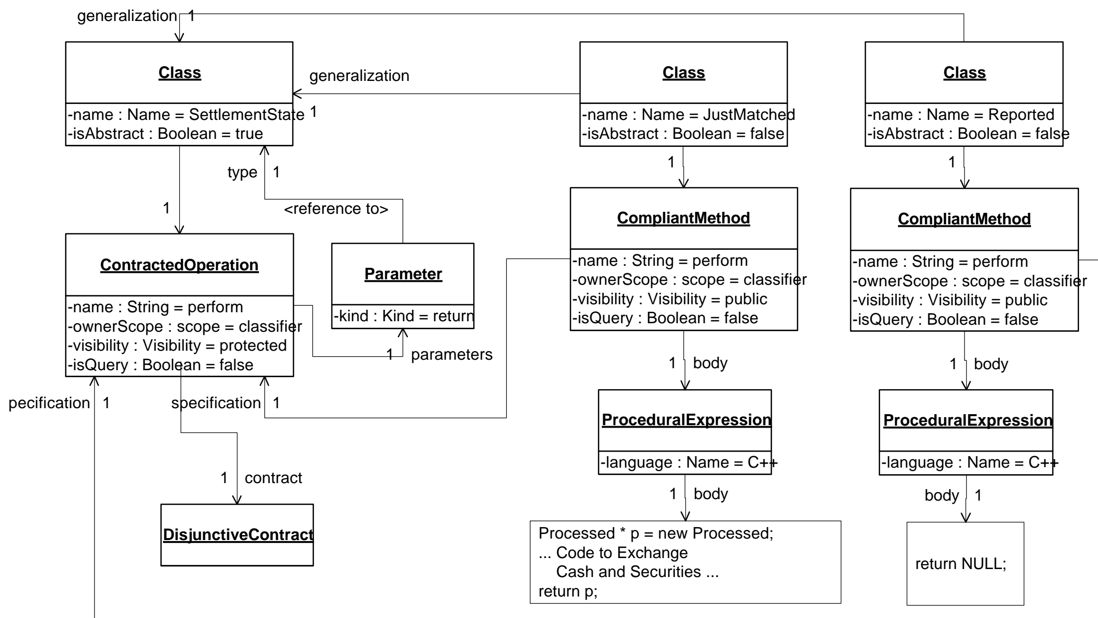

Figure 7. Adding the Actual Settlement State Classes

JustMatched and Reported are each shown in Figure 7 to provide a single CompliantMethod (perform, of course), whose body is presented in C++, and whose specification is shown to be the perform ContractedOperation of our base SettlementState class. The implementation of perform by JustMatched creates and returns a new Processed (a sub-class of SettlementState) object.and thus conforms to the alternateContractA of the perform ContractedOperation depicted in Figure 6. Whereas, the implementation of perform by Reported returns NULL, and thus complies with alternateContractB. Since these implementations of perform are contract-complaint, both will be allowed to execute, which is exactly what we want. If we had attempted, however, to add the an ill-behaved class, whose perform method simply returned a pointer to the object

it was invoked on, its perform method would not have been able to be executed, since the whole protection mechanism we have just built up prevents this.

5. Two-level modeling

Figures 6 and 7 clearly capture a rich and expressive model; by appealing to the abstract syntax we have been able to express things which were simply inexpressible in the standard UML concrete syntax. However, Figures 6 and 7 are obviously rather verbose. Modeling using only abstract syntax, then, is long-winded and tedious work. Reading such models is equally cumbersome. We would like to retain some of the succinctness of the concrete syntax, without losing the expressive power of the (extensible) abstract syntax. To respond to this problem we have derived a concept that we term two-level modeling.

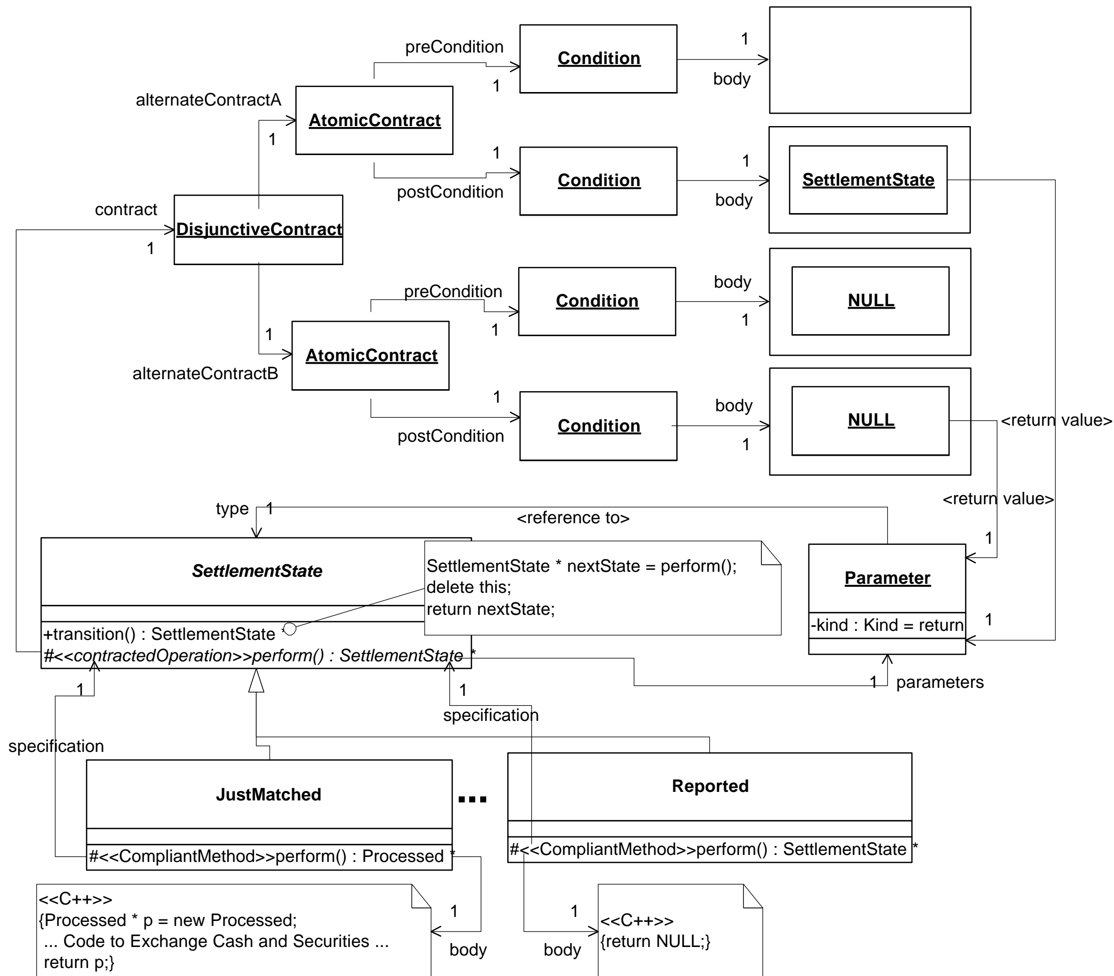

Figure 8. A Two-Level Model of Settlement State

The basic realization underlying two-level modeling is that different fragments of a given model place different expressibility requirements on the modeling language. In our settlement system example, UML concrete syntax is perfectly adequate for much of our model. The perform operation, on the other hand, requires direct appeal to the meta-model. To force the wholesale expression of SettlementState and its derived classes in the verbose abstract syntax just to accommodate the requirements of perform is clearly overkill. A more appealing strategy is to use the concrete syntax wherever possible, and only appeal to the

meta-model (i.e. the abstract syntax) where essential. We term this strategy two-level modeling, since it mixes two modeling levels - the traditional concrete syntax level, and the meta-model. In principle, this strategy could appeal to ever higher levels such as the meta-meta-level, which defines the abstract syntax and semantics for the meta-model, and so on. Two-level modeling is generalizable, therefore, into a multi-level modeling strategy. For our example, however, two-level modeling is perfectly adequate and so in Figure 8 we re-express the model captured in Figures 6 and 7, using a blend of concrete and abstract syntax.

The two most striking things about Figure 8, compared with Figures 6 and 7, are probably its compactness and its greater readability. Much of the specification for the classes SettlementState, JustMatched, and Reported is presented using standard UML concrete syntax. There is, however, appeal to the meta-level: the perform operation of SettlementState is shown to have a contract and a return parameter both captured using abstract syntax. The JustMatched and Reported classes are shown to have perform methods whose specification is provided by the perform operation, and each method's body is shown in C++ code. In summary, Figure 8 captures the same model as Figures 6 and 7, but two-level modeling has, we believe, resulted in a far more appealing realization.

6. Conclusions and further work

In this paper we have proposed two-level modeling as a promising strategy for gaining the expressiveness of abstract syntax without completely sacrificing the succinctness of concrete notation. We are beginning to see for ourselves that this whole approach seems to have quite broad applicability. For example, we have discovered that the specification of design patterns [1] is often improved via a two-level modeling strategy [5]. We have now started to investigate the idea of extending the concrete syntax of the modeling language, allowing the association of new concrete graphical icons with our abstract syntax extensions. This should enable our models to be expressed in an even more compact, readable, and intuitive form. We term these concrete icons *visual stereotypes*. As our research with visual stereotypes advances, we anticipate its subsequent documentation.

7. References

[1] E. Gamma, R. Helm, R. Johnson, and Vlissides.J. *Design Patterns: Elements of Reusable Object-Oriented Software* , Addison-Wesely, 1995.

[2] UML Consortium. *Unified Modeling Language 1.1, Notation Guide*, www.omg.org, 1997.

[3] UML Consortium. *Object Constraint Language*, www.omg.org, 1998.

[4] UML Consortium. *Unified Modeling Language 1.1, Semantics Guide*, www.omg.org, 1997.

[5] A. Lauder and S. Kent. Precise Visual Specification of Design Patterns. In: *Proceedings of ECOOP '98*, ed. E. Jul. Springer-Verlag, 1998.