HKU CS Tech Report TR-95-04

NOODLE++: a 3-dimensional Net-based Object-Oriented Development Model [1]

T.H. Tse
Department of Computer Science
The University of Hong Kong
Hong Kong

C.P. Cheng
Department of Computer Science
The University of Hong Kong
Hong Kong

Abstract

Object-oriented analysis and design methodologies are considered as the most popular software development methods for the 1990s. A common drawback, however, is that they have been developed informally. There is no theoretical framework enabling us to define precisely the object-oriented concepts involved, to solve concurrency problems, and to verify the correctness of the implementation.

We propose a 3-dimensional net structure behind object-oriented software development. This structure consolidates the concepts of classes, inheritance, overloading and message passing through a single model. Inheritance and overloading can be unified with message passing in a 3-dimensional representation, but are physically differentiable by occupying the vertical and horizontal planes, respectively, of the model.

Furthermore, the formal model can be mapped to various object-oriented analysis and design notations. The theoretical framework can thus be adopted for systems validation and verification for existing methodologies. The liveness and consistencies of objects can be verified, and inheritance and interaction coupling can be identified and checked.

1 Introduction

Object-oriented analysis and design methodologies [1, 2, 3, 4, 5, 6] are considered as the most popular software development methods for the 1990s. Their main selling point is a user-friendly front-end graphical interface. A common drawback, however, is that they have been developed informally. The target systems cannot be defined precisely and implementations cannot be validated and verified in the absence of a supporting formal framework.

On the other hand, a number of formal object-oriented specification languages [7, 8, 9, 10, 11] have been developed, independently of the popular methodologies. Practitioners are, however, rather reluctant to use these formal tools since unfamiliar languages are involved [12, 13].

To bridge the gap between object-oriented graphical notations and formal languages, we propose a 3-dimensional net-based object-oriented development model (NOODLE++). [2] We provide formal syntax and semantics to informal object-oriented concepts such as classes, inheritance, overloading and message passing.

Our model has both graphical and algebraic representations which are semantically equivalent. The graphical representation can be used for visualization. Furthermore, unlike conventional diagrams, the graphical representation of NOODLE++ is 3-dimensional. This provides an additional degree of freedom towards integrating different aspects of a target system, but allows software engineers to project complex graphics into 2-dimensional views for the ease of understanding.

Our model can be mapped to popular object-oriented methodologies such as object-oriented analysis (OOA) and object-oriented design (OOD) of Coad and Yourdon [2, 3]. Thus software engineers can still manipulate their designs using a preferred notation. In this way, the algebraic representation can be adopted for systems validation and verification for existing methodologies.

---

[1] This research is supported in part by a grant of the Research Grants Council.

[2] The acronym NOODLE was introduced in our earlier paper [14] to stand for ''a 3-dimensional Net-based Object-Oriented DeveLopment Environment''. We suggest reading ''NOODLE++'' as ''noodles'' to avoid a mouthful.

2 Advantages of using Net Theory

We have chosen predicate/transition nets, a type of high-level Petri net, to model the main features in object-oriented software development. The reasons for our choice can be summarized as follows:

(*a*) An object-oriented software system consists of a number of objects which run persistently and concurrently, depending on the mutual interactions among one another. Net theory is particularly useful for modelling interactions among distributed components.

(*b*) Net theory also provides an algebraic syntax and semantics for modelling the internal structures and behaviours of individual objects.

(*c*) Besides the algebraic representation, nets have an equivalent graphical representation, which resembles the pictorial aspect of popular object-oriented analysis and design notations, making the theory more acceptable to software engineers who are used to graphical methods.

(*d*) Simple graphical tools are not sufficiently powerful for the description of real-life software systems. A collection of tools, such as object diagrams, state-transition diagrams and data flow diagrams, have to be used together to model the structural and dynamic aspects of complex systems. This may lead to inconsistencies, incompleteness and ambiguities. Since net theory covers both the structural and dynamic properties, there is no need for a second modelling tool, hence alleviating the problems of consistency.

(*e*) Data and operations are given equal emphasis in object-oriented software development, but most of the graphical tools can only model one aspect. For example, data flow diagrams show the transitions between processes, and state-transition diagrams show the transitions between states of data. Nets, on the other hand, are a bipartite graphical language which models data and processes with the same level of importance.

(*f*) Unlike structured development methods, which emphasize only top-down design, object-oriented development also supports bottom-up techniques. In net theory, the concepts of net refinement and abstraction can support both top-down and bottom-up techniques.

(*g*) Class reuse is one of the features of object-orientedness. This can be supported by embedding reusable subnets to newly defined nets.

(*h*) The instantiation of objects requires specific class definitions to be extracted from a class hierarchy. This can be supported by the concept of sectioning in net theory.

3 The NOODLE++ Model

3.1 Classes

We specify a class by an extended predicate/transition net [15], which has both graphical and algebraic representations with equivalent semantics. For example, the graphical representation of the class *Account* is shown in Figure 1. A method such as *credit* is specified as a transition, which is denoted by a bar. It is connected to incoming message paths through which client objects such as *Customer* send in their requests. It may also have return message paths so that results are returned to the appropriate client objects after execution. It reads and updates attributes such as *Balance*. Messages and attributes are specified as predicates, which are denoted by spheres.

Transitions and predicates are linked by arcs denoting message paths, specified by an input function $\bullet E$ and an output function $E\bullet$. The logical relationships E define the logical relationships among the message paths. In our example, the input and output functions of *credit* are

$$\bullet E(credit) = \{Balance, In_{credit}\}$$
$$E\bullet(credit) = \{Balance, Out_{credit}\}$$

and the respective logical functions are

$$\bullet E(credit) = Balance \textbf{ and } In_{credit}$$
$$E\bullet(credit) = Balance \textbf{ and } Out_{credit}.$$

The structure of a message or an attribute is specified on each arc by an algebraic term defined over the signature (D, Σ_D). An incoming message consists not only of its own contents, but also the identity of client object, so that a return message can be sent when necessary. We define a special operation **id** in Σ_D, so that an object d is represented by a term **id**(d). A logic operator **and** is used to connect the identity to the message content. In our example, the contents of the incoming messages are

$\bullet D_{credit}(Balance) = balance$
$\bullet D_{credit}(In_{credit}) = \mathbf{id}(Customer)$ **and** *amount*

and the contents of the outgoing messages are

$D\bullet_{credit}(Balance) = balance'$
$D\bullet_{credit}(Out_{credit}) = \mathbf{id}(Customer)$.

In addition, incoming messages may originate from one of many possible classes of client objects. Such messages are denoted by a set of incoming message arcs, each of which represents messages from each class. These arcs could be related by the **and** operator (which is represented by ''$*$'' in the graphical representation) and an **xor** operator (which denotes exclusive-or and is represented graphically by ''+'').

For return messages, instead of using the **xor** operator, another logic operator **sel** is adopted, denoting ''selection'' and is represented graphically by ''♦''. The **xor** operator is used for incoming messages whose structures are decided by the client objects. However, **xor** exhibits a non-deterministic property and, if used in return messages, would not enable us to decide the recipient of the return message. The **sel** operator, on the other hand, uniquely identifies the client object as a function of the contents of the message.

Thus we formally define a class $c = (N, E, D)$ as follows:

(*a*) $N = (P, E, \bullet E, E\bullet)$ is a predicate/transition net such that

- (*i*) $P = A \cup B$, where A is a non-empty set of attributes and B is a non-empty set of messages.
- (*ii*) E is a non-empty set of methods.
- (*iii*) $\bullet E$: $E \rightarrow 2^P \setminus \{\varnothing\}$ is known as the input function. It specifies the arcs flowing towards each method in E. For any method $e \in E$, $(\bullet E)(e)$ is known as the pre-set of e.
- (*iv*) $E\bullet$: $E \rightarrow 2^P \setminus \{\varnothing\}$ is known as the output function. It specifies the arcs flowing from each method in E. For any method $e \in E$, $(E\bullet)(e)$ is known as the post-set of e.

Note that 2^P denotes the power set of P.

(*b*) $E = (T_{\Sigma_P}, \bullet\mathbf{E}, \mathbf{E}\bullet)$ specifies the logical relationships among the arcs, such that

- (*i*) $\Sigma_P = \{$**and**, **xor**, **sel**$\}$ is the set of logic operators on P, and T_{Σ_P} is the term algebra over the signature (P, Σ_P).
- (*ii*) $\bullet\mathbf{E}$: $E \rightarrow \mathbf{S} \setminus \{\varnothing\}$ is the input logic function, where $\mathbf{S} \subseteq T_{\Sigma_P}$ is the term algebra over the signature $((\bullet E)[E], \{$**and**, **xor**$\})$.
- (*iii*) $\mathbf{E}\bullet$: $E \rightarrow \mathbf{T} \setminus \{\varnothing\}$ is the output logic function, where $\mathbf{T} \subseteq T_{\Sigma_P}$ is the term algebra over the signature $((E\bullet)[E], \{$**and**, **sel**$\})$.

(*c*) $D = (T_{\Sigma_D}, \bullet\mathbf{D}, \mathbf{D}\bullet)$ specifies the terms on the arcs, such that

- (*i*) D is the set of message parameters and attribute values, $\Sigma_D = \{$**id**, **and**, **xor**, **sel**$\}$ is the set of operators on D, and T_{Σ_D} is the term algebra over the signature (D, Σ_D).
- (*ii*) $\bullet\mathbf{D}$ is the family $(\bullet\mathbf{D}_e\text{: } (\bullet E)(e) \rightarrow T_{\Sigma_D(X)} \setminus \{\varnothing\})_{e \in E}$ of input term functions, where $T_{\Sigma_D(X)}$ is the term algebra over the signature (D, Σ_D) and the set X of message parameter variables.
- (*iii*) $\mathbf{D}\bullet$ is the family $(\mathbf{D}_e\bullet\text{: } (E\bullet)(e) \rightarrow T_{\Sigma_D(X)} \setminus \{\varnothing\})_{e \in E}$ of output term functions.

3.2 Inheritance and Overloading

Inheritance is a relationship between two classes b and c such that c acquires the structure and behaviours of b. Thus, the class c makes use of the methods and attributes of the class b and provides services similar to those of b. b is known as the superclass of c, and c the subclass of b. Furthermore, inheritance supports overloading in the sense that some methods and attributes of class b can be redefined in class c and thus provide services which have the same name but perform differently.

The relationship between superclass and subclass is modelled by linking up the two corresponding nets to become a single, 3-dimensional net. In the resulting net, the top and bottom layers represent the superclass and subclass, respectively. This is illustrated by the *ChequeAccount*, *OverdraftAccount* and *SavingsAccount* examples as shown in Figures 2 to 4. Formally, we define an inheritance hierarchy as a six-tuple $(C, L, \bullet C, C\bullet, \bullet\mathbf{C}, \mathbf{C}\bullet)$, where:

(*a*) C is the set of classes in the hierarchy.
(*b*) L is the set of predicates which link up the classes.

(*c*) $\bullet C$: $C \rightarrow 2^C$ is the input function of the classes.
(*d*) $C\bullet$: $C \rightarrow 2^C$ is the output function of the classes.
(*e*) $\bullet$**C** is the input logic function of the classes, mapping each class to a set of predicates over the **xor** operator.
(*f*) **C**$\bullet$ is the output logic function of the classes, mapping each class to a set of predicates over the **sel** operator.

The **xor** operator is used to amalgamate the external input arcs of each method in the superclass, including passive incoming message arcs and inheritance links, so that the messages sent from either the superclass or the subclass can activate the method. On the other hand, the **sel** operator is used to amalgamate the external output arcs of each method in the superclass, including passive return message arcs and inheritance links, so that the destination of the return messages can be determined according to whether the incoming messages originated from the superclass or the subclass.

Further details of inheritance will be given in the following sections.

3.2.1 Inheritance of Methods and Attributes

Suppose $e_b \in E_b$ is a method in class b to be inherited by class c. Predicates, transitions, and arcs will have to be added to c for this purpose. Moreover, links have to be set up between b and c by which the inheritance hierarchy is built. This is illustrated by the *ChequeAccount* example in Figure 2. The new transition *credit* of *ChequeAccount* transfers incoming messages from the predicate In_{credit} to the original transition *credit* of *Account*, where actual processing takes place. Outgoing messages are then transferred back to *credit* of *ChequeAccount* for further transmission to the outside world via the predicate Out_{credit}. In general terms,

(*a*) Suppose $e_b \in E_b$ is a method in class b which only accepts incoming messages through the predicate p_{e_b}. Suppose e_b is inherited by class c. The responsibility of the new transition $e_c \in E_c$ is no more than transferring incoming messages from the subclass c to the superclass b. The transition e_b is where actual actions take place. Thus, e_c is linked to the outside world by a predicate p_{e_c} and to e_b by a predicate i_{e_c}.

  (*i*) A transition e_c, responsible for accepting and redirecting incoming messages, is added to E_c. It is shown immediately below e_b in the graphical representation.

  (*ii*) A message predicate p_{e_c}, responsible for incoming messages, is added to $(\bullet E)(e_c)$. (Intuitively, this means that there will be an arc connecting the predicate p_{e_c} to e_c.) The predicate is shown immediately below p_{e_b} in the graphical representation.

  (*iii*) An inheritance predicate i_{e_c} is added to $(C\bullet)(c)$ and $(\bullet C)(b)$. (Intuitively, this means that there will be arcs linking e_c through the predicate i_{e_c} to e_b.) The predicate is shown between e_c and e_b in the graphical representation.

(*b*) Suppose $e_b \in E_b$ is a method in class b which not only accepts incoming messages through the predicate p_{e_b}, but also returns messages through the predicate q_{e_b}. Thus, e_c is linked to the outside world by input and output predicates p_{e_c} and q_{e_c}, and similarly to e_b by input and output inheritance predicates i_{e_c} and o_{e_c}. In this case, apart from (*i*), (*ii*) and (*iii*) above, we have:

  (*iv*) An output inheritance predicate o_{e_c} is added to $(C\bullet)(b)$ and $(\bullet C)(c)$. (Intuitively, this means that there will be arcs linking e_b through the predicate o_{e_c} to e_c.) The predicate is shown between e_b and e_c in the graphical representation.

  (*v*) A message predicate q_{e_c}, responsible for return messages, is added to $(\bullet E)(e_c)$. The predicate is shown immediately below q_{e_b} in the graphical representation.

Attributes can also be inherited by subclasses. A subclass c inheriting an attribute a_b from a superclass b means that a newly defined method e_c in c uses the a_b defined in b. To model the inheritance of attributes, NOODLE++ allows methods in the subclass to access attributes in the superclass directly. Algebraically, the inheritance hierarchy is augmented with a set of predicates between the subclass and the superclass, similarly to the case of the inheritance of methods.

3.2.2 Overloading of Methods and Attributes

A method can have one set of actions in a superclass b and another set in a subclass c. We say that the method is overloaded. We can also say that the method in b is overridden by another method of the same name in c. This is illustrated by the *OverdraftAccount* example in Figure 3. The new transition *debit* of *OverdraftAccount* replaces the original transition *debit* of *Account*. The former is shown immediately below the latter in the graphical representation. Unlike the previous case of normal inheritance, however, *debit* of *OverdraftAccount* will not pass messages to *debit* of *Account*, but will process *Balance* directly.

In general terms, let $e_b \in E_b$ be a method defined in class b. Suppose it is to be overridden by a method $e_c \in E_c$ defined in class c. A transition e_c, together with the corresponding message paths, must be added to subclass c. Instead of calling the method e_b as Section 3.2.1, however, e_c accesses the corresponding attributes in b directly. In other words, e_c will be directly connected to the predicates corresponding to the attributes in b.

Like the inheritance of methods, two categories of overloaded methods must be considered:

(*a*) Suppose $e_b \in E_b$ is a method in class b which only accepts incoming messages through the predicate p_{e_b}. Suppose e_b is overridden by a method e_c in class c.

- (*i*) A transition e_c is added to E_c. It is shown immediately below e_b in the graphical representation.
- (*ii*) A message predicate p_{e_c}, responsible for incoming messages, is added to $(\bullet E)(e_c)$. (Intuitively, this means that there will be an arc connecting the predicate p_{e_c} to e_c.) The predicate is shown immediately below p_{e_b} in the graphical representation.
- (*iii*) For any attribute $a_b \in A_b$ inherited by c, a_b is added to $(E\bullet)(e_c)$ and $(\bullet E)(e_c)$. (Intuitively, this means that there will be request and return arcs connecting the attribute a_b to e_c.)

(*b*) Suppose $e_b \in E_b$ is a method in class b which not only accepts incoming messages through the predicate p_{e_b}, but also returns messages through the predicate q_{e_b}. In this case, apart from (*i*), (*ii*) and (*iii*) above, we have:

- (*iv*) A message predicate q_{e_c}, responsible for return messages, is added to $(\bullet E)(e_c)$. The predicate is shown immediately below q_{e_b} in the graphical representation.

Attributes can also be overridden by subclasses. The modelling of overloading of attributes is similar to the above and will not be discussed separately.

3.2.3 Newly Defined Methods and Attributes

Besides inheriting and overloading methods and attributes from a superclass, new methods and attributes can also be defined for subclasses. To define a new method e_c, a new transition e_c is added to the subclass layer. This is illustrated by the transition *calInterest* in the *SavingsAccount* example in Figure 4. We note that no other transition should be immediately above it in the superclass layer. We note also that the new method may access original attributes such as *Balance* defined in the superclass. Algebraically, a transition e_c will be inserted into E_c and other predicates will be added as appropriate.

Similarly, a new attribute a_c' can be defined for the class c by introducing a new predicate a_c' in the subclass layer c, together with the appropriate connections and corresponding algebraic modifications.

3.2.4 Definition of a Subclass

The construction of inheritance, overloading and new attributes and methods are in fact part of an integrated NOODLE++ model of class hierarchies. Visualization is made simple by showing only the components on selected 2-dimensional planes. This can be achieved by projecting a NOODLE++ graphical representation from the bottom to the top, as shown in the bottom layers of Figures 2 to 4. The resulting 2-dimensional graphics will be identical to that produced from the definition for a single class.

Multiple inheritance can also be represented in a similar fashion.

4. Applications of NOODLE++

A formal specification provides a framework for software engineers to specify, develop, and verify systems in a systematic way that cannot be achieved by informal specifications [16]. In this section, we further illustrate how this advantage of formal methods is made possible by NOODLE++. On one hand, our model

can be mapped to popular object-oriented analysis and design methodologies such as Coad and Yourdon, thus providing a theoretical framework for validating and verifying the structural and dynamic properties. For the purpose of this basic function, the formal framework can be transparent to users. On the other hand, the 3-dimensional graphical representation in NOODLE++ can also provide sophisticated users with a means of deeply understanding the infrastructure of the target system with a view to improving the design.

4.1 Applications of a 3-Dimensional Model

4.1.1 Unification of Inheritance and Message Passing

A class *c* may wish to make use of an existing method in another class *b*. This can be achieved either by inheriting *b* or by passing a message to *b*. During the analysis phase of object-oriented development, software engineers often have difficulty in deciding between inheritance relationships and message passing connections, and may wish to change their minds later in the design phase.

In most object-oriented development methodologies, inheritance and message passing are entirely distinct concepts so that it is almost impossible to revert from one decision to the other afterwards. In NOODLE++, we provide a single unifying framework behind them. Through a CASE tool with a window environment, software engineers can change an inheritance connection to a message passing connection simply by dragging the superclass to the same level as the subclass.

This is illustrated in the example shown in Figure 5. Suppose that a class *Stack* is going to be implemented, and another class *List* is already available. We are probably tempted to inherit *Stack* from *List*, as in many examples in the formal literature. Thus, pushing an element on to the stack can be achieved by adding an element to the head of the list while popping an element from a stack corresponds to removing an element from the head of the list. If we do so, however, unwarranted *List* operations that add or remove elements at arbitrary positions will also be inherited. If these are ever used, the class *Stack* will not behave properly. Should we have any regret on our decision, we can change the inheritance link into a message passing link by dragging the corresponding elements in NOODLE++.

4.1.2 Projection of the 3-Dimensional Model

Although we live in a world in which physical objects are 3-dimensional, we must admit that we may not be smart enough to visualize all of them. We often translate 3-dimensional objects into a set of 2-dimensional views in our mind for the ease of understanding.

We face a similar problem in the 3-dimensional graphical representation in NOODLE++. In view of this, the supporting graphical display provides 2-dimensional views through projection. For example, a projection from the bottom to the top of the 3-dimensional representation would result in a simple class definition, as illustrated in Figures 2 to 4. A projection with compression from the side would result in a simple class hierarchy diagram, as illustrated in Figure 6. The projections are supported by the corresponding algebraic representations.

4.2 Validation and Verification Support for Existing Object-Oriented Methods

4.2.1 Mapping to Existing Object-Oriented Methods

We appreciate that practicing software engineers are reluctant to use newly invented notations with which they are not familiar. Although NOODLE++ supports a graphical representation which is semantically equivalent to the algebraic formalism, it is not meant to replace other object-oriented analysis and design notations. The model can be mapped to existing object-oriented analysis and design notations. For example, typical links between NOODLE++ and Coad/Yourdon notations can be summarized as follows:

NOODLE++	Coad and Yourdon
Attribute predicates	Attributes in the attribute layer
Method transitions	Services in the service layer
Input parameter predicates	Input parameters in service chart specifications
Output parameter predicates	Output results in service chart specifications
Functions defined by arc and method refinements	Details in object state diagrams and service charts

In this way, we can indirectly provide software engineers with formal syntax and semantics to existing object-oriented notations, so that we can improve the quality of the design by various means of validation and verification as suggested in the following sections.

4.2.2 Verification of Object States

Objects are persistent entities having states that change with the course of events. Hence object-oriented analysis methodologies support the modelling of states and their transitions. Most of the methods, however, do not support the verification of correctness of the states, since they do not have the formal framework for the purpose. For instance, states and transitions are described by object-state diagrams and service charts of Coad and Yourdon, but deadlock and livelock situations may remain unidentified.

Reachability trees in NOODLE++, on the other hand, enable impossible states to be detected easily. An algorithm for producing reachability trees for the purpose has been adapted from [17]. Thus, reachable states can be identified through the traversal of the reachability tree. Given an initial state of an object, the following are verified:

(*a*) All valid states are reachable.
(*b*) All reachable states are valid.

It has been shown [17] that the number of predicates in predicate/transition nets, by which NOODLE++ is modelled, is remarkably less than the number of nodes in conventional graphical models such as Petri nets. Thus the technique can be applicable to real-life systems.

4.2.3 Consistency of an Object

Many object-oriented analysis notations, such as state-transition diagrams and data flow diagrams [5], support multi-level specification of objects and methods. It would be most useful if consistencies between levels could be verified. Following our earlier work [18], we subdivide consistency checks into two subcategories, namely structural and behavioural consistencies. Most CASE tools provide a limited degree of support over structural consistencies only. Even so, they are based on *ad hoc* techniques so that the checks are not guaranteed to be complete.

Suppose a transition e is refined in a multi-level diagram into a subnet N. We introduce the concept of external input and output terms, which are input and output messages not consumed by other methods in the subnet. The decomposition of e into N will be structurally consistent if and only if

(*a*) $(\bullet \mathbf{D}_e)[(\bullet E)(e)]$ = the set of external input terms of N and
(*b*) $(\mathbf{D}_e \bullet)[(E \bullet)(e)]$ = the set of external output terms of N.

Behavioural consistency refers to whether the dynamic properties of an object or method are preserved by the subnet. This concept is distinctly absent in most CASE tools because of the corresponding absence of the reachability concept. In NOODLE++, we verify the behavioural consistency by determining whether, for a given external input state (known as marking in net theory), the external output states (or markings) of both e and its subnet N agree.

The algorithms for determining external input/output terms and states are similar to those in [18] and will not be reproduced here.

4.2.4 Checking of Inheritance Coupling

After an inheritance relationship has been established between two classes, a method or an attribute of the subclass may be:

(*a*) Inherited from the superclass.
(*b*) Overloading the corresponding method or attribute in the superclass.
(*c*) A new method or attribute.

As suggested by Coad and Yourdon, high inheritance coupling is desirable, which means that as many methods and attributes of the superclass should be inherited by the subclass as possible.

There is no easy way, but manually, to check inheritance coupling in the common object-oriented development methodologies. We can, however, check the inheritance coupling in NOODLE++ through the pre- and post-sets of the methods and attributes in the subclass.

4.2.5 Checking of Interaction Coupling

As opposed to inheritance coupling, low interaction coupling is desirable in object-oriented software systems. In other words, we should keep the number of connections for message passing between objects to a minimum. For example, it is undesirable to have message pass-throughs, which refer to messages passed from object *b* through *c* to *d* such that no attempt is made by *c* to change the message contents.

Similarly to inheritance coupling, interaction coupling in common object-oriented analysis notations can only be checked manually. NOODLE++ supports the checking of interaction coupling through the pre- and post-sets of the methods.

5 Conclusion

We have proposed NOODLE++, a 3-dimensional net-based object-oriented development model, in terms of extended predicate/transition nets. We have formally defined the fundamental object-oriented concepts, such as classes, inheritance, overloading and message passing.

NOODLE++ has both graphical and algebraic representations which are semantically equivalent. The graphical representation defines classes and message passing in a 2-dimensional view as well as inheritance and overloading in a 3-dimensional view. Inheritance and overloading can thus be unified with message passing, but are physically differentiable by occupying the vertical and horizontal planes, respectively, of the model. With the aid of a 3-dimensional software package, the model can be manipulated and visualized easily by projecting on to selected 2-dimensional planes.

NOODLE++, however, is not meant to be yet another object-oriented methodology. The model can be mapped into existing object-oriented methodologies such as OOA and OOD of Coad and Yourdon. The algebraic representation helps to provide popular object-oriented notations with formal syntax and semantics so that the validation and verification of target systems can be achieved. The liveness and consistencies of objects can be verified, and inheritance and interaction coupling can be identified and checked.

References

1. Booch G. Object-oriented analysis and design with applications. Benjamin/Cummings, Redwood City, California, 1994
2. Coad P, Yourdon E. Object-oriented analysis. Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1991
3. Coad P, Yourdon E. Object-oriented design. Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1991
4. Embley DW, Kurtz BD, Woodfield SN. Object-oriented systems analysis: a model driven approach. Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey, 1992
5. Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. Object-oriented modeling and design. Prentice-Hall, Englewood Cliffs, New Jersey, 1991
6. Wirfs-Brock RJ, Wilkerson B, Wiener L. Designing object-oriented software. Prentice-Hall, Englewood Cliffs, New Jersey, 1990
7. Carrington DA, Duke D, Duke R, King P, Rose G, Smith G. Object-Z: an object-oriented extension to Z. In: ST Vuong (ed) Formal description techniques II: Proceedings of 2nd IFIP WG 6.1 international conference (FORTE '89). North-Holland, Amsterdam, 1990, pp 281–296
8. Cusack, E, Lai M. Object-oriented specification in LOTOS and Z (or my cat really is object-oriented!). In: de Bakker JW, de Roever WP, Rozenberg G (eds) Foundations of object-oriented languages: Proceedings of school/workshop on research and education in concurrent systems (REX). Springer-Verlag, Berlin, 1991, pp 179–202 (Lecture Notes in Computer Science, Vol. 489)
9. Durr EHH, van Katwijk J. VDM++: a formal specification language for object-oriented designs. In: Proceedings of 6th IEEE international conference on computer systems and software engineering (CompEuro '92). IEEE Computer Society, Los Alamitos, California, 1992, pp 214–219
10. Alencar AJ, Goguen JA. OOZE: an object-oriented Z environment. In: America P (ed) Object-oriented programming: Proceedings of 5th European conference (ECOOP '91). Springer-Verlag, Berlin, 1991, pp 180–199 (Lecture Notes in Computer Science, Vol. 512)
11. Lakos CA, Keen CD. LOOPN++: a new language for object-oriented Petri nets. In: Proceedings of 1994 European simulation multiconference. North-Holland, Amsterdam, 1994
12. Tse TH. A unifying framework for structured analysis and design models: an approach using initial algebra semantics and category theory. Cambridge University Press, Cambridge, 1991 (Cambridge Tracts in Theoretical Computer Science, Vol. 11)

13. Tse TH. Formal or informal, practical or impractical: towards integrating formal methods with informal practices in software engineering education. In: Barta B-Z et al (eds) Software engineering education: Proceedings of IFIP WG 3.4 working conference. North-Holland, Amsterdam, 1993, pp 189–197
14. Tse TH, Cheng CP. Towards a 3-dimensional net-based object-oriented development environment (NOODLE). 5th international congress on computational and applied mathematics (ICCAM '92), Leuven, Belgium, 1992 (Also Technical Report TR-92-05, Department of Computer Science, The University of Hong Kong, Hong Kong, 1992)
15. Reisig W. Petri nets: an introduction. Springer-Verlag, Berlin, 1985 (EATCS Monographs on Theoretical Computer Science, Vol. 4)
16. Wing JM. A specifier's introduction to formal methods. IEEE Computer 1990; 23 (9): 8–24
17. Huber P, Jensen AM, Jepsen LO, Jensen K. Reachability trees for high-level Petri nets. Theoretical Computer Science 1986; 45 (3): 261–292
18. Tse TH, Pong L. Towards a formal foundation for DeMarco data flow diagrams. The Computer Journal 1989; 32 (1): 1–12

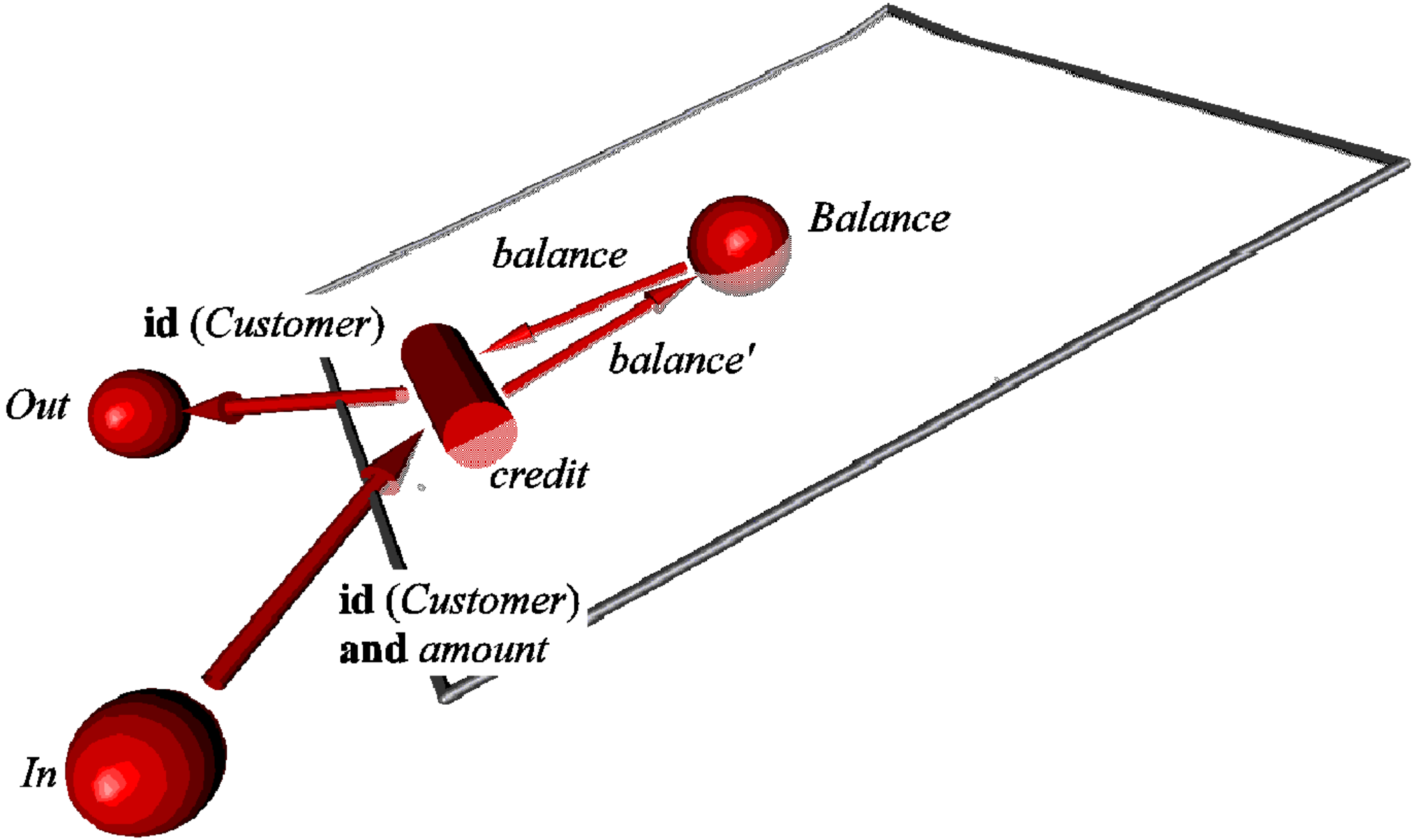

Figure 1 Class

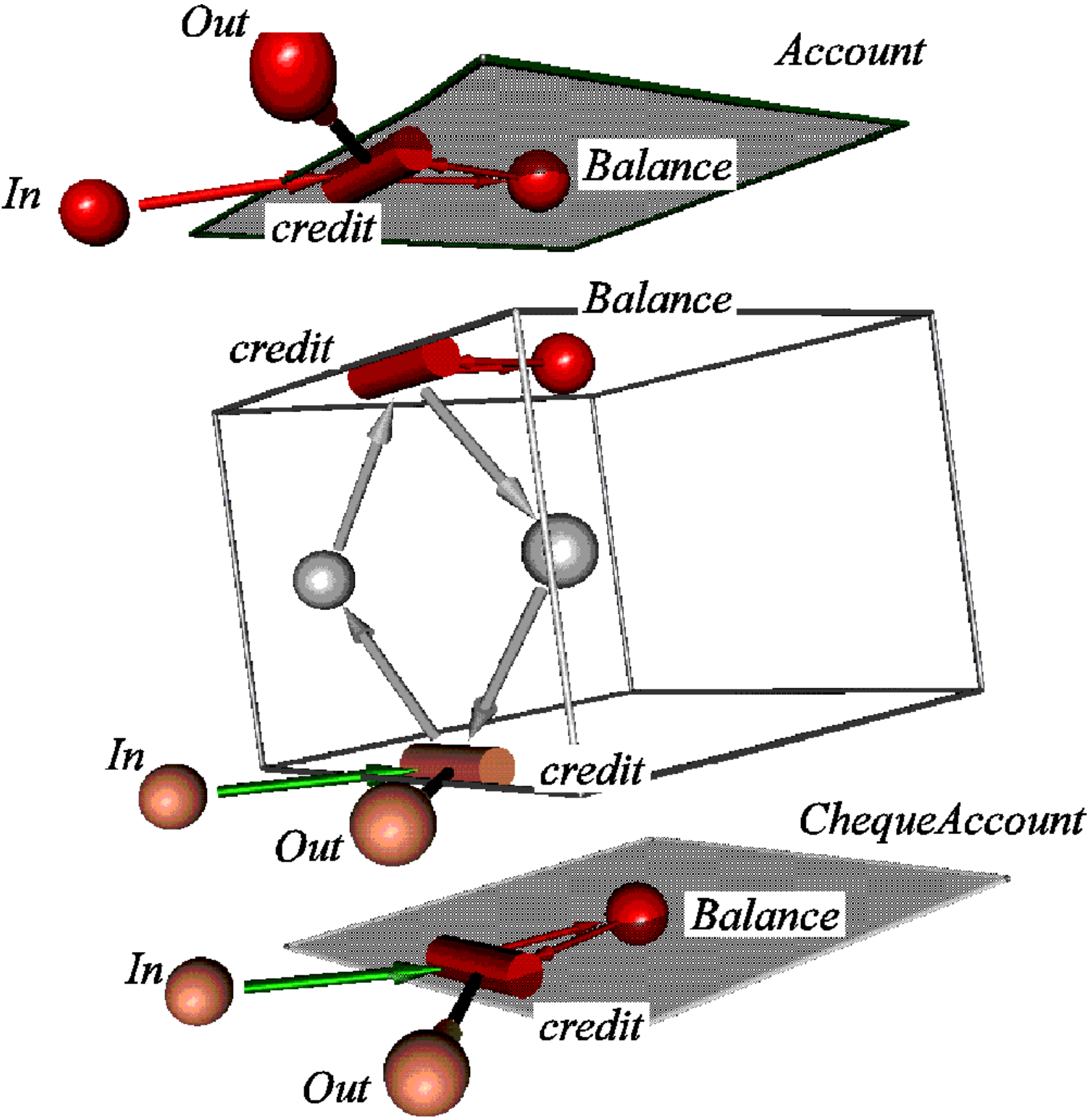

Figure 2 Inheritance of Method

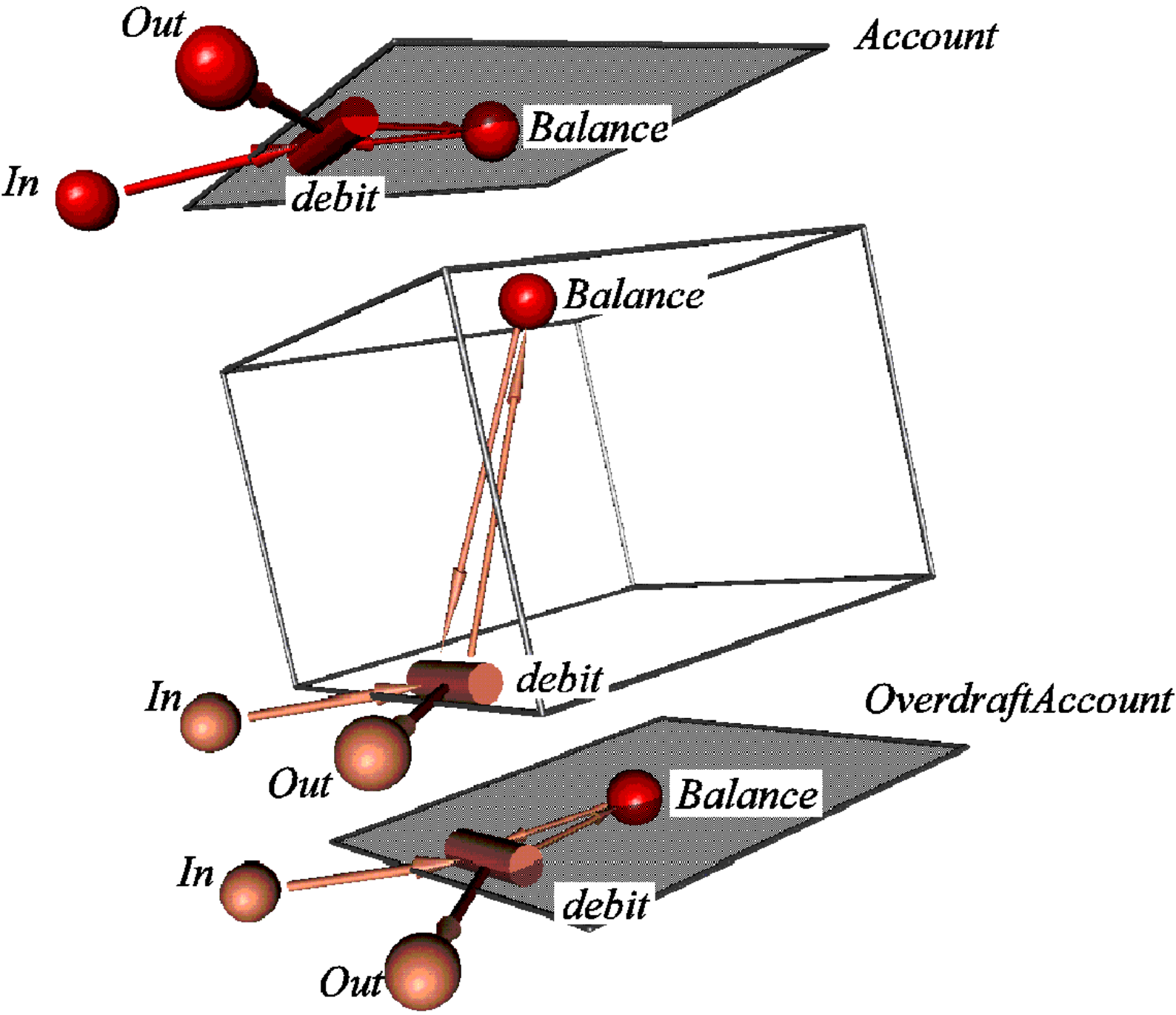

Figure 3 Overriding of Method

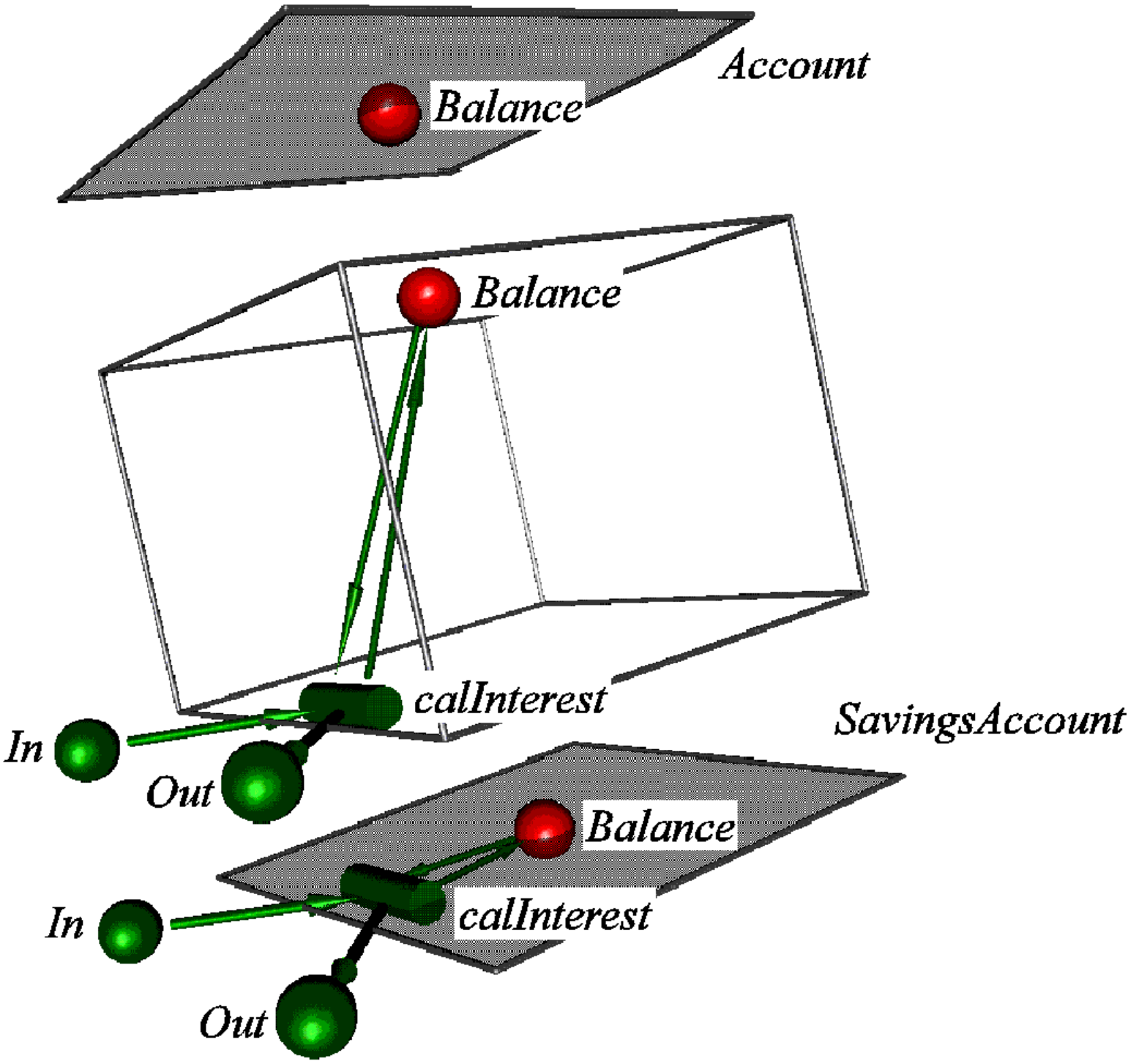

Figure 4 Newly Defined Method

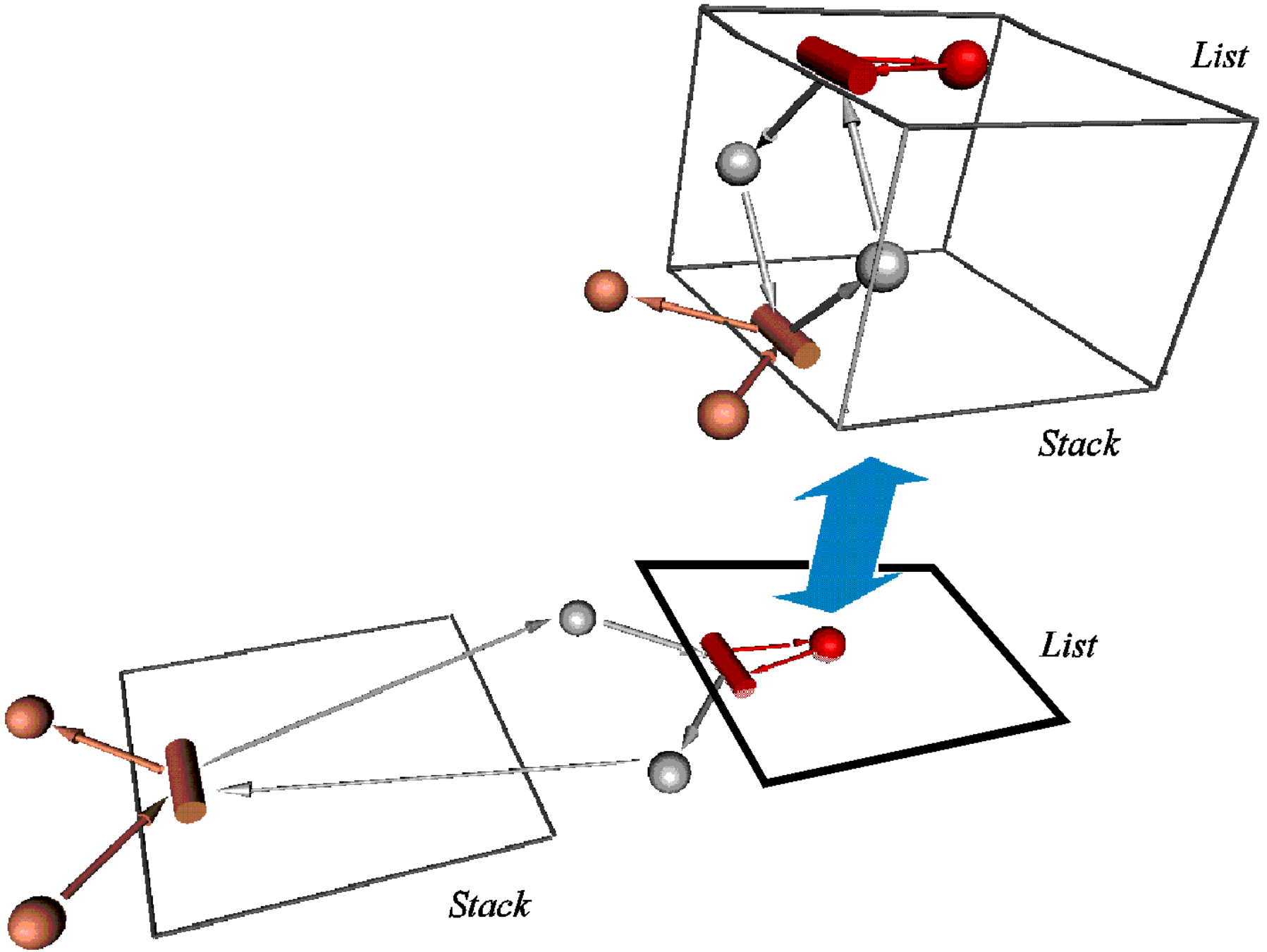

Figure 5 Unification of Inheritance and Message Passing

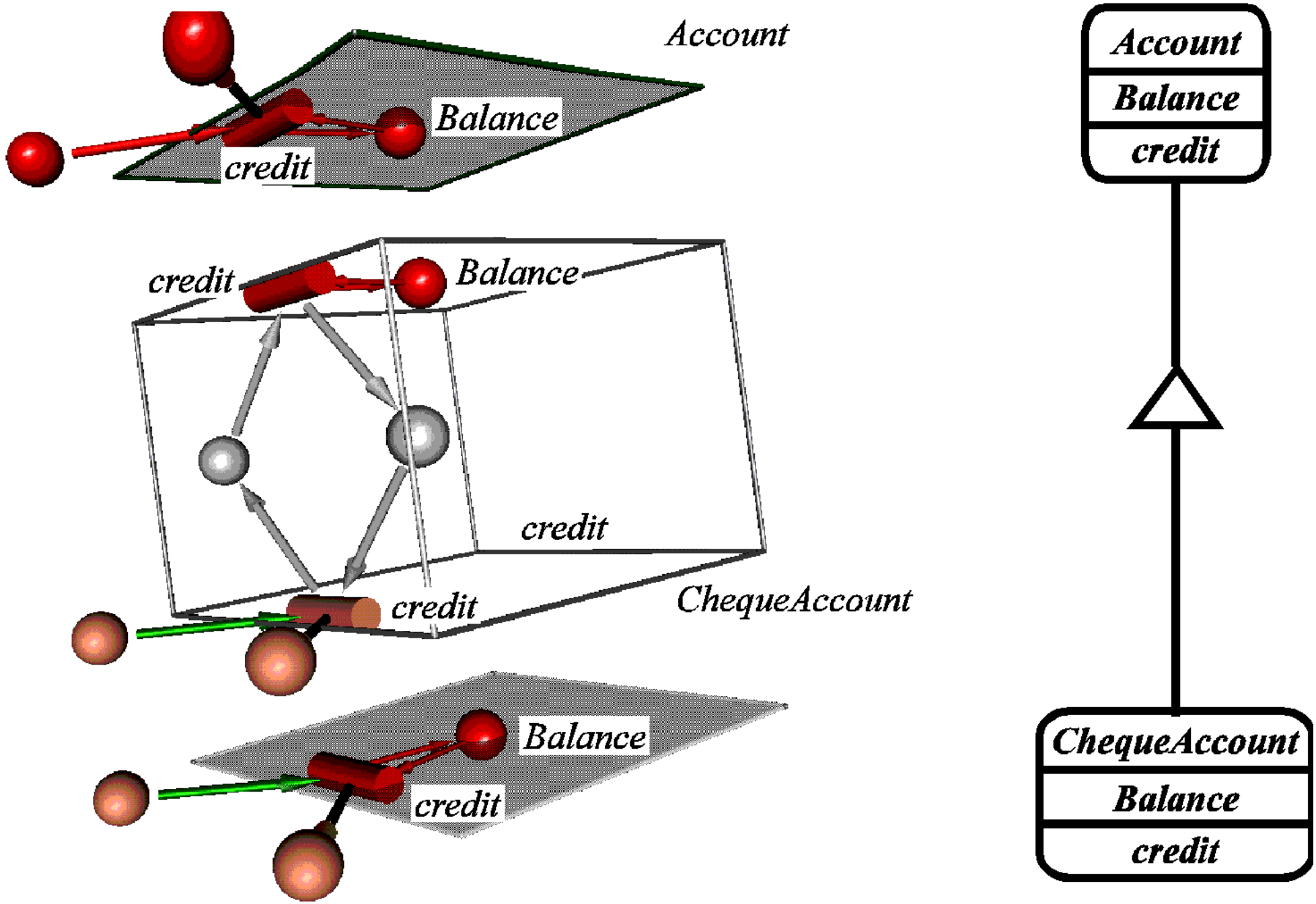

Figure 6 Projections of 3-Dimension Model