FOOD: a Graphical Interface for Object-Oriented Algebraic Specifications *

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

Tel: (+852) 2859 2183
Fax: (+852) 2559 8447
Internet: tse@cs.hku.hk

ABSTRACT

Functional Object-Oriented Design (FOOD) attempts to provide a bridge between the popular graphics-based object-oriented methods that have no theoretical foundation, and the less popular algebraic specification languages that have a reliable mathematical base. We propose a set of graphical interface for Functional Object-Oriented Programming System (FOOPS), a formal object-oriented specification language with algebraic semantics. The algebraic declarations of classes, methods, attributes and inheritance are specified in the notions of data flow diagrams, state diagrams, and object diagrams. The algebraic axioms defining the behavioral properties of objects are specified in terms of state diagrams, data flow diagrams and object structure charts. The user-friendly graphical interface enables software engineers to accept algebraic specifications much more easily, while retaining the advantages of the underlying formal semantics.

Keywords: Formal specifications, object-oriented specifications, algebraic specifications

1. INTRODUCTION

Two conflicting schools of thought have dominated software engineering.

(a) Popular Methods

One school stresses on the popular software development methodologies such as object-oriented analysis and design. These methods are based on the experience of outstanding consultants in the industry, and are supported by CASE tools with user-friendly graphical interfaces. Examples are the Booch method [1], Object Modeling Technique (OMT) [2], the Unified Method [3], Object-Oriented Analysis (OOA) [4], and Object-Oriented Design (OOD) [5].

A common drawback among many of the popular development methods is that they have been developed informally and are not supported by any formal syntax or semantics. There is no means to guarantee either the consistency of a specification or a smooth transition to the implementation phase of software development. Most of the guidelines suggest iterations of undos and redos when the outcome is not satisfactory. CASE tools for supporting the methodologies are developed in an ''evolutionary'' approach because method designers are not aware of any fundamental problems in the tools until they are tried in real life and bugs

---

are found. Horror stories on poorly designed systems using popular methodologies are not uncommon.

(b) Formal Methods

The other school advocates formal methods that help to specify unambiguously the exact requirements of software systems. Examples are the use of algebraic semantics such as in OBJ3 [6, 7] and FOOPS [8, 9], abstract models such as in Z [10], and concurrency models such as in CSP [11]. Because of the formal syntax and semantics behind the specifications, programs can be developed with a higher degree of reliability. The advantages of using formal methods include the following:

(*i*) The specifications will not be ambiguous or inconsistent. It can be verified to be syntactically correct and semantically consistent. Misunderstandings can be avoided.

(*ii*) The implementation can be proved mathematically to be correct. Using formal derivations, we can predict the behavior of the programs independently of how they are coded, and verify that the implementation indeed agrees with what is required. This is particularly essential to systems that are safety-critical, or where heavy financial loss would result from errors.

(*iii*) Despite popular misconceptions, the time to develop the system can be reduced despite a slightly longer time for specification. (See [12], for example.) It will also be easier to make future enhancements for the system.

On the other hand, even though outcries have been made to popularize the formal methods, they have largely remained proposals from the academia. Practicing software engineers are not too eager to apply them. Students trained in formalism often find themselves having to learn and use informal methodologies after graduation. This may be attributed to the following:

(*i*) A specification should serve also as a means of communication among users, designers and implementors. It helps people to arrive at the requirements of the target system. It enables users to visualize the proposal and compare it with their actual needs. For example, the popular methodologies are mostly based on graphical documentation that are more suitable for giving users an intuitive idea of complex systems. It is in two dimensions, may be color-coded and may consist of multiple levels. Hence it provides presenters with additional degrees of freedom. Formal methods do not support these functions very well.

(*ii*) On the contrary, a formal specification consists of a substantial amount of jargon that have to be translated verbally by the designer into laymen terms before it could be understood by end users. This is in sharp contrast to other engineering disciplines where blueprints presented to users consist of diagrams instead of differential equations.

(*iii*) A user of a large complex system would not be familiar with all of its operations but only interested in an overview plus details of some selected aspects. Thus it is more natural to approach a system in a top-down manner, starting from a high level of abstraction and then filling in the details of the lower levels. Most formal methods, however, require us to define the details at the beginning and grow a specification bottom-up.

(*iv*) Most practising software engineers are not trained in formal methods, and may be overwhelmed simply by the thought of having to use mathematical notations.

Instead of debating on which school to follow, we must recognize that neither formal nor informal methods are a complete solution to software development. Our Software Engineering Group have successfully conducted a number of projects in bridging the gap between popular

methodologies and various formal methods, such as algebraic semantics and category theory [13], net theory [14], process algebra [15], and logic programming [16]. We have joint projects with the universities of Oxford, York, Melbourne, and Jinan, and awarded research fundings of 3.7 million dollars from various international sources. We have been recognized internationally as the pioneer in methods integration. Further examples of projects by other researchers to bridge the gap between formal and informal methods may be found in [17, 18, 19, 20].

In this paper, we report on the project known as Functional Object-Oriented Design (FOOD). We propose a graphical front-end interface for Functional Object-Oriented Programming System (FOOPS) [8, 9] that has a formal algebraic semantics based on OBJ3 [6, 7]. Most of the graphical notations in FOOD are adapted from existing methodologies, thus relieving the user from having to learn an alien set of mathematical symbols. The algebraic declarations of classes, methods, attributes, and inheritance are specified in the notations of data flow diagrams, state diagrams, and object diagrams. The algebraic axioms that define the behavioral properties of the objects are specified in terms of state diagrams, data flow diagrams, and object structure charts. Practitioners need not learn the concept of axioms in algebraic specifications. They do not even see an equal sign. User training and experience on existing methodologies, such structured analysis or object-oriented analysis will not be wasted. On the contrary, they would be very useful in supporting FOOD.

We assume that readers are familiar with data flow diagrams, state diagrams, object diagrams, and structure charts that are commonly used in objected-oriented modeling and structured analysis and design. Interested readers may refer to [2, 21, 22] for details.

2. EXAMPLE OF A FOOPS SPECIFICATION

Figure 1 shows an example of a FOOPS specification of three modules, namely an account, a savings account and a checking account. Each module is identified by the module name preceded by the keyword `omod`. The line ''`protecting ACCOUNT`'' indicates that another module `ACCOUNT` is imported and used in the current module. The line ''`SavingsAccount < Account`'' indicates that `SavingsAccount` is a subclass of `Account`. The methods (preceded by ''`me`'') and the attributes (preceded by ''`at`'') are declared in the form of mathematical functions. A method such as `credit` changes the state of an object. An attribute in FOOPS is a query that returns the value of a property of an object without changing any state of the object. For example, ''`balance`'' returns the current balance of a specific account. The behaviors of such methods and attributes are defined by axioms that are either equations or conditional equations. Suppose, for example, we `credit` an account A by an amount `M`. The new balance will be the same as the old balance plus `M`. This is written as an equational axiom as follows:

```
ax balance(credit(A, M)) = balance(A) + M .
```

Suppose, further, that we `debit` an account A by an amount `M`. If sufficient funds are available, then the new balance will be the same as the old balance minus `M`. This is written as a conditional equation as follows:

```
cax balance(debit(A, M)) = balance(A) - M
    if balance(A) >= M .
```

FOOPS support all the object-oriented concepts, including objects, classes, attributes, methods, inheritance, polymorphism, private (hidden) classes and methods, and generic modules through parameterization. The axioms are given a formal meaning through an algebraic semantics [23], a reflective semantics [8], a sheaf semantics [24], and an operational semantics [25].

```
omod ACCOUNT is

protecting TRANSACTION-HISTORY .
class Account .

at balance : Account -> Money [default: (0)] .
at transactionHistory : Account -> TransactionHistory
   [default: (<<>>)] .
me credit : Account Money -> Account .
me debit : Account Money -> Account .
me transfer : Account Account Money -> Account .

var A : Account .
var A' : Account .
var M : Money .

ax balance(credit(A, M)) = balance(A) + M .

ax transactionHistory(credit(A, M))
   = transactionHistory(A) << today ; M >> .

cax balance(debit(A, M)) = balance(A) - M if balance(A) >= M .

cax transactionHistory(debit(A, M))
    = transactionHistory(A) << today ; - M >>
    if balance(A) >= M .

cax transfer(A, A', M) = debit(A, M) ; credit(A', M)
    if balance(A) >= M .

endo *** ACCOUNT


omod SAVINGS-ACCOUNT is

protecting ACCOUNT .
class SavingsAccount .
subclass SavingsAccount < Account .

fn rate : -> Float .
me interest : SavingsAccount -> SavingsAccount .

var A : SavingsAccount .

ax rate = 0.1 .
ax interest(A) = credit(A, balance(A) * rate) .

endo *** SAVINGS-ACCOUNT
```

Figure 1 Example of a FOOPS Specification

```
omod CHECKING-ACCOUNT is

protecting SAVINGS-ACCOUNT .
protecting QUEUE-OF-CHECK .
class CheckingAccount .
subclass CheckingAccount < Account .

at savingsOf : CheckingAccount -> SavingsAccount .
me linkSavings : CheckingAccount SavingsAccount
    -> CheckingAccount .
me honorCheck : CheckingAccount Check -> CheckingAccount .
me processCheck : CheckingQueue CheckingAccount
    -> CheckingQueue .

var C : Check .
var CQ : CheckingQueue .
var CA : CheckingAccount .
var SA : SavingsAccount .

ax savingsOf(linkSavings(CA, SA)) = SA .

cax honorCheck(CA, C) = debit(CA, amount(C))
    if balance(CA) >= amount(C) .

cax honorCheck(CA, C)
    = transfer(savingsOf(CA), CA, amount(C) - balance(CA)) ;
      debit(CA, amount(C))
    if balance(CA) < amount(C) .

cax processCheck(CQ, CA)
    = honorCheck(CA, front(CQ)) ;
      removeCheck(CQ)
    if | CQ | == 1 .

cax processCheck(CQ, CA)
    = honorCheck(CA, front(CQ)) ;
      removeCheck(CQ) ;
      processCheck(CQ, CA)
    if | CQ | > 1 .
endo *** CHECKING-ACCOUNT
```

Figure 1 (continued)

3. THE FOOD INTERFACE

The following is a brief outline of the procedure as recommended by the FOOD method:

(*i*)   Identify objects.

(*ii*)   Identify the relationships among objects with the help of object diagrams.

(*iii*) Look for common attributes and methods, and transitive dependence. They may imply associations, aggregations, or inheritance.

(*iv*)   Normalize the relationships, paying special attention to the meaningfulness of the objects rather than the convenience of implementation.

(*v*)   Draw data flow diagrams with nested classes. Convert them into FOOPS declarations of classes, attributes, methods, and subclasses.

(*vi*)   For each attribute, list its states and connect them into state diagram(s). Convert them into attribute-related axioms.

(*vii*) For each method, expand them into smaller components using multi-level data flow diagrams. Convert them into method-related axioms.

(*viii*) If the life history of an object involves sequentiality control, draw an object structure chart. Convert them into sequentiality-related axioms.

Details of the FOOD interface are as follows:

3.1 DECLARING CLASSES, METHODS, ATTRIBUTES, AND INHERITANCE

We express the methods and attributes of given objects using data flow diagrams [2, 21], as shown in Figure 2. Objects are denoted by double bars, and data items by arrows. Methods and attributes are denoted by bubbles, and distinguished as follows:

(*a*) A method bubble is always connected via an input-and-output arrow to a class. There may be any number of input arrows (possibly none) that are labeled by data types or connected to other classes. Examples are `credit` and `open` in Figure 2(*a*).

(*b*) An attribute bubble is always attached to an output arrow labeled by a data type, and is always connected to at least one class via an input arrow. There may be any number of input arrows (possibly none) that are labeled by other data types. An example is `balance` in Figure 2(*b*).

Inheritance between superclasses and subclasses can be described in the form of a nested diagram shown in Figure 3. The popular nesting notation in state diagrams [2] and the standard inheritance symbol in object diagrams [2] are adopted.


3.2 DEFINING ALGEBRAIC AXIOMS

3.2.1 Defining Attribute-Based Axioms

State diagrams [2, 21] model the events that affect the attributes of an object. Instead of forcing users to ''think equations'', we need only ask them to consider how methods will change the attributes. For example, consider the attribute `balance` of `CheckingAccount`, as shown in Figure 4(*a*). This can be translated into attribute-related axioms in FOOPS:

```
var A : Account .
var M : Money .
```

(a) Classes and Methods

Money
→ credit ↖
                  ▭▭ Account ▭▭
Customer ▭▭ → open ↙
         ▭▭
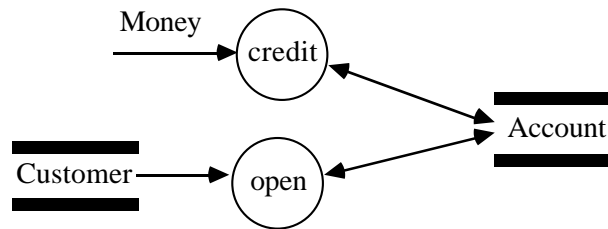
(b) Classes and Attributes

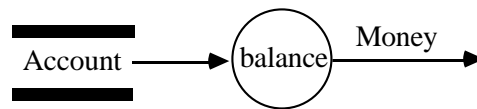▭▭ Account ▭▭ → balance Money →

Figure 2   Declaring Classes, Methods, and Attributes
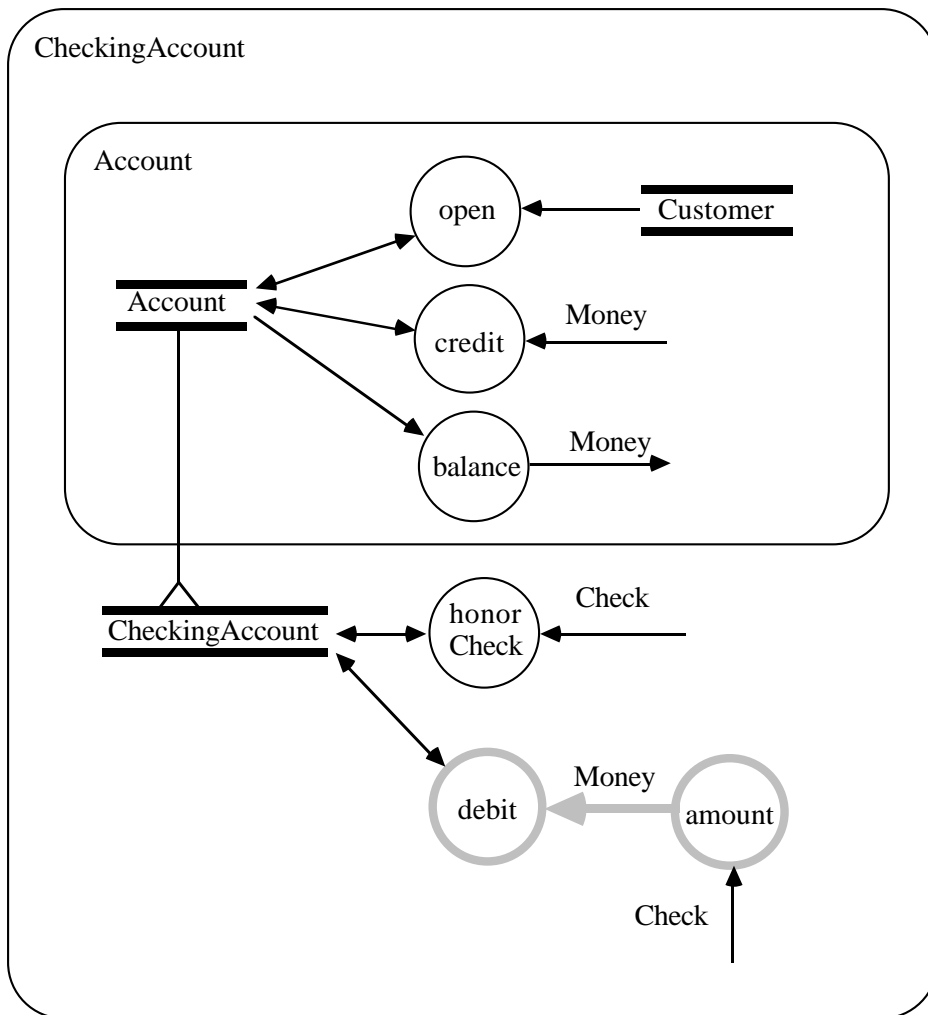
Figure 3   Declaring Inheritance

```
ax balance(credit(A, M)) = balance(A) + M .
ax transactionHistory(credit(A, M))
   = transactionHistory(A) << today ; M >> .
```

Conditions can also be modeled in state diagrams. An example is shown in Figure 4(*b*). Such conditions can be translated into conditional axioms in FOOPS, thus:

```
var A : Account .
var M : Money .

cax balance(debit(A, M)) = balance(A) - M
    if balance(A) >= M .
cax transactionHistory(debit(A, M))
    = transactionHistory(A) << today ; - M >>
    if balance(A) >= M .
```

3.2.2 Defining Method-Based Axioms

When preparing data flow diagrams, we may start with a context diagram that shows only the whole object as one bubble. We then expand it into a number of second level bubbles, thus providing more details to the users. Each of the second level bubbles may be further expanded. This may be carried on in a top down fashion.

Each expansion process can be translated into a method-related axiom in FOOPS. We shall illustrate the expansion of methods using several common features in data flow diagrams:

(a) Data Couples

Data couples are the most common feature in data flow diagrams. A data couple connection means that two bubbles are linked by the passage of data. For example, the bubble honorCheck in Figure 5(*a*) is expanded into a sequence of two bubbles, amount and debit, linked by a data couple Money. The FOOD diagram can be translated into FOOPS in a straightforward manner, resulting in the following code:

```
var C : Check .
var CA : CheckingAccount .

ax honorCheck(CA, C) = debit(CA, amount(C)) .
```

(b) Control Couples

Two bubbles X and Y are control coupled if and only if the execution of one bubble must be fully completed before the other could start, but no actual data is passed from one to another. For example, when transferring money from SavingsAccount to CheckingAccount, a typical bank would do the debit first, and wait to see if it is successful before starting the credit process. This situation is drawn in Figure 5(*b*). It can be translated into FOOPS as follows:

```
var SA : SavingsAccount .
var CA : CheckingAccount .
var M : Money .

ax transfer(SA, CA, M)
   = debit(SA, M) ; credit(CA, M) .
```
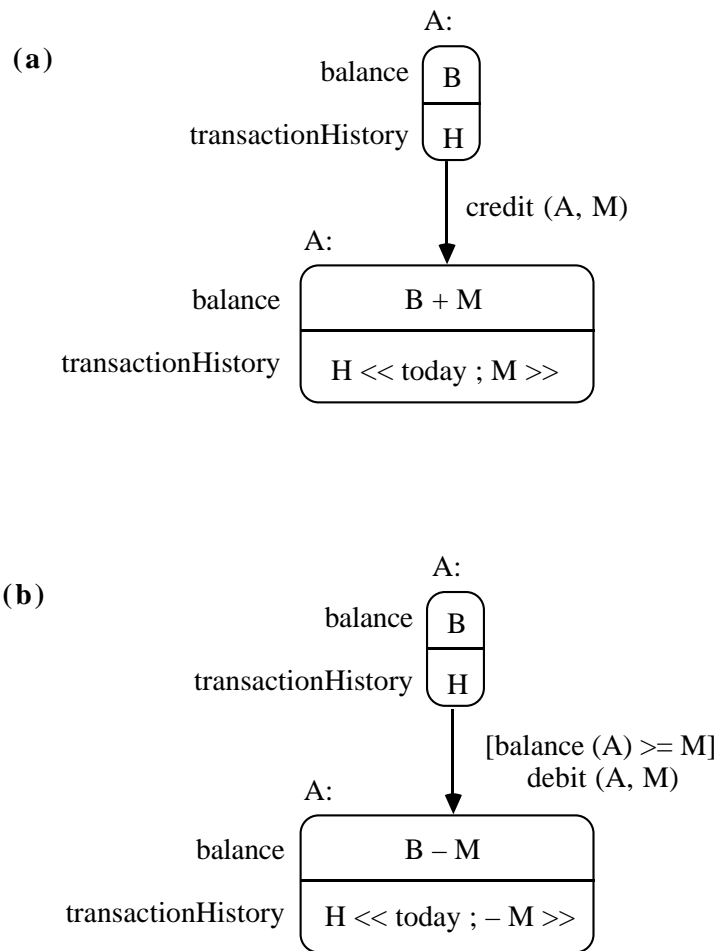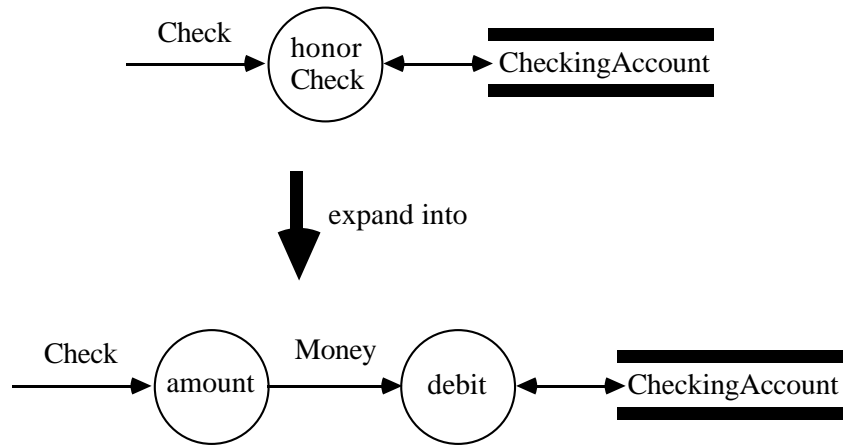
(a)

A:

balance B

transactionHistory H

credit (A, M)

A:

balance B + M

transactionHistory H << today ; M >>

(b)

A:

balance B

transactionHistory H

[balance (A) >= M]
debit (A, M)

A:

balance B – M

transactionHistory H << today ; – M >>

Figure 4   Defining Attribute-Based Axioms

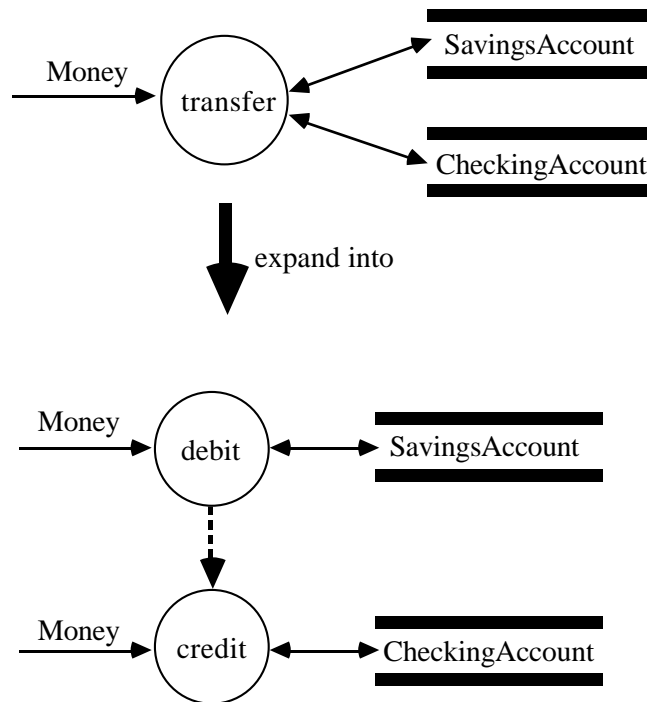**(a) Data Couples**

(b) Control Couples

Figure 5   Defining Method-Based Axioms

(c) Concurrent Data Couples

Two bubbles in a data flow diagram may not be directly related to one another, but a third bubble may need data from each of them before it can be executed. We call them concurrent data couples. An example is shown in Figure 5(*c*), where the method `honorCheck` is expanded into four bubbles: two functions `amount` and `bankCharge` that may take place simultaneously, following by a function "`_+_`" and a method `debit`. Translation into FOOPS will again be straightforward:

```
var C : Check .
var CA : CheckingAccount .

ax honorCheck(CA, C)
    = debit(CA, amount(C) + bankCharge(C)) .
```

3.2.3 Defining Sequentiality-Based Axioms

In some complex situations, simple data couples or control couples may not be sufficient to express a method. More specific sequentiality controls need to be expressed. An object structure chart, adapted from structure charts in structured design [22], will be employed for this purpose. It helps us understand a more comprehensive life history of an object. As an illustration, consider the processing of a queue of checks for a specific account. It would consist of simple iterations of

(*a*) obtaining a check from the front of the queue
(*b*) honoring the check by debiting from the checking account.
(*c*) transferring money from the savings account if there is not sufficient funds.
(*d*) removing the check.

This procedure can be specified easily as a structure chart as shown in Figure 6, which may then be translated automatically into sequentiality-related axioms in FOOPS:

```
var C : Check .
var CQ : CheckingQueue .
var CA : CheckingAccount .

cax processCheck(CQ, CA)
    = honorCheck(CA, front(CQ)) ;
      removeCheck(CQ)
    if | CQ | == 1 .

cax processCheck(CQ, CA)
    = honorCheck(CA, front(CQ)) ;
      removeCheck(CQ) ;
      processCheck(CQ, CA)
    if | CQ | > 1 .

cax honorCheck(CA, C) = debit(CA, amount(C))
    if balance(CA) >= amount(C) .

cax honorCheck(CA, C)
    = transfer(savingsOf(CA), CA,
                amount(C) - balance(CA)) ;
      debit(CA, amount(C))
    if balance(CA) < amount(C) .
```
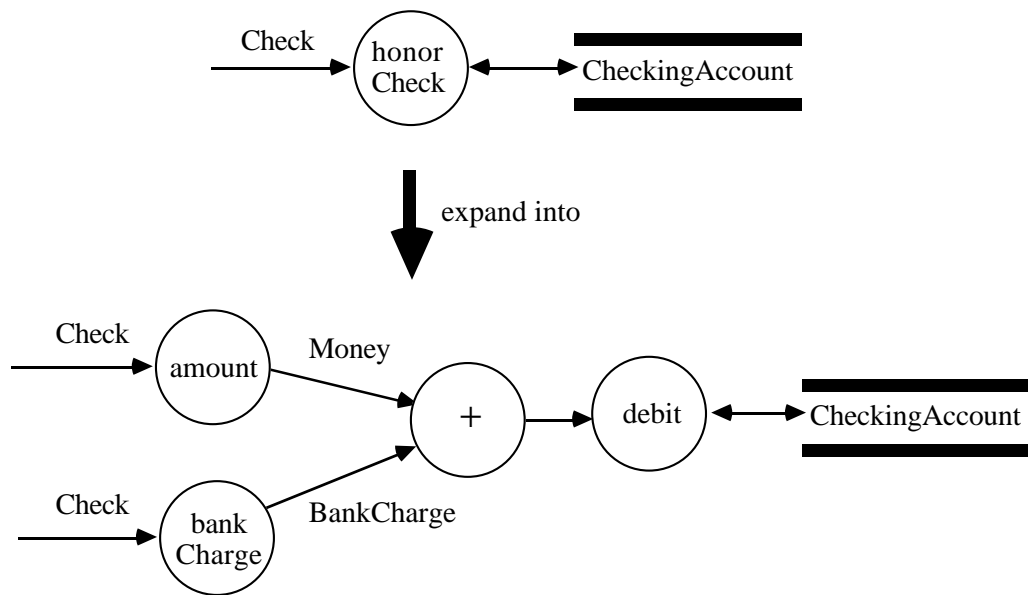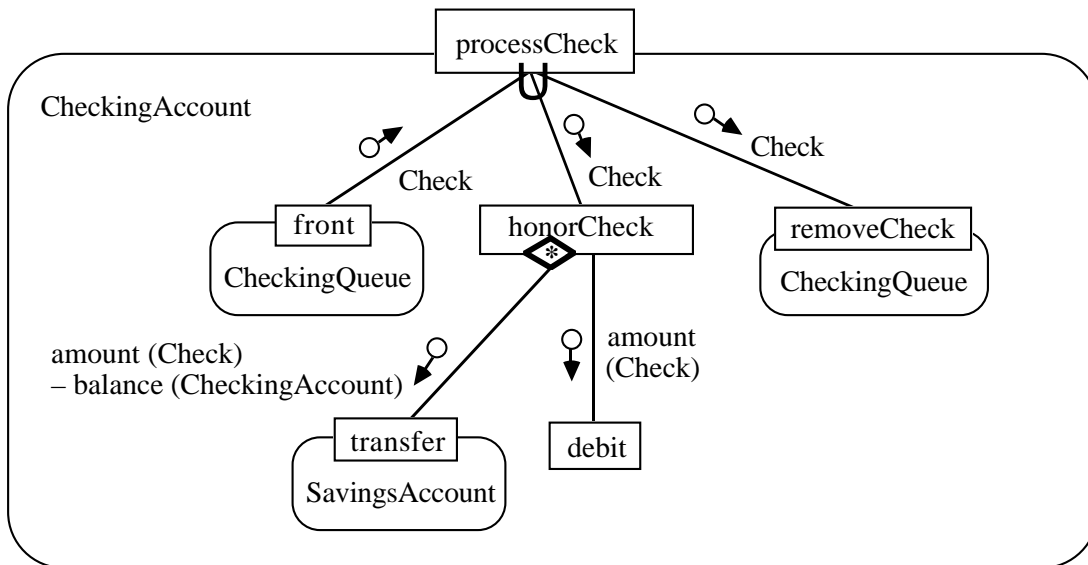
(c) Concurrent Data Couples

Figure 5 **(Continued)**

* amount (Check) > balance (CheckingAccount)

Figure 6 Defining Sequentiality-Based Axioms

4. CONCLUSION

FOOD is an attempt to provide a bridge between object-oriented algebraic specifications and popular graphical notations. We propose a set of graphical interface for FOOPS. The algebraic declarations of classes, methods, attributes, and inheritance are specified in the notations of data flow diagrams, state diagrams, and object diagrams. The algebraic axioms that define the behavioral properties of the objects are specified in terms of state diagrams, data flow diagrams, and object structure charts. The user-friendly graphical interface enables software engineers to accept algebraic specifications much more easily, while retaining the advantages of the underlying formal semantics.

We are currently implementing FOOD on a windows-based system, and testing the feasibility of the FOOD method on realistic case studies such as its application in a manufacturing system [26].

REFERENCES

[1] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, California (1994).

[2] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey (1991).

[3] G. Booch and J. Rumbaugh, *Unified Method for Object-Oriented Development: Documentation Set*, Rational Software Corporation, Santa Clara, California (1995).

[4] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1991).

[5] P. Coad and E. Yourdon, *Object-Oriented Design*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1991).

[6] J.A. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, and J. Meseguer, ''An introduction to OBJ3'', in *Conditional Term Rewriting Systems: Proceedings of 1st Workshop*, S. Kaplan and J.-P. Jouannaud (eds.), Lecture Notes in Computer Science, Vol. 308, Springer-Verlag, Berlin, pp. 258–263 (1988).

[7] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, ''Introducing OBJ'', in *Applications of Algebraic Specification using OBJ*, J.A. Goguen, D. Coleman, and R.M. Gallimore (eds.), Cambridge University Press, Cambridge (1995).

[8] J.A. Goguen and J. Meseguer, ''Unifying functional, object-oriented, and relational programming with logical semantics'', in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (eds.), MIT Press, Cambridge, Massachusetts, pp. 417–477 (1987).

[9] L. Rapanotti and A. Socorro, ''Introducing FOOPS'', Technical Report PRG-TR-28-92, Programming Research Group, Oxford University Computing Laboratory, Oxford (1992).

[10] J.C.P. Woodcock, *Using Standard Z: Specification, Proof and Refinement*, Prentice-Hall International Series in Computer Science, Prentice-Hall, Hemel Hempstead, Hertfordshire, UK (1994).

[11] J. Davies, G.M. Reed, A.W. Roscoe, and S.A. Schneider, *Advanced CSP*, Prentice-Hall International Series in Computer Science, Prentice-Hall, Hemel Hempstead, Hertfordshire, UK (1995).

[12] G. Jones, ''Sharp as a razor: a Queen's award for the Computing Laboratory'', *Oxford Magazine* (59) (1990).

[13] T.H. Tse, *A Unifying Framework for Structured Analysis and Design Models: an Approach using Initial Algebra Semantics and Category Theory*, Cambridge Tracts in Theoretical Computer Science, Vol. 11, Cambridge University Press, Cambridge (1991).

[14] T.H. Tse and C.P. Cheng, ''NOODLE++: a 3-dimensional net-based object-oriented development model'', in *Proceedings of 2nd Methods Integration Workshop*, A. Bryant *et al.* (eds.), Springer-Verlag, Berlin (1996).

[15] W.K. Chan, *Inheritance and OMT: a CSP Approach*, M.Phil. Thesis, The University of Hong Kong, Hong Kong (1995).

[16] T.H. Tse, T.Y. Chen, F.T. Chan, H.Y. Chen, and H.L. Xie, ''The application of Prolog to structured design'', *Software: Practice and Experience* **24** (7): 659–676 (1994).

[17] R.H. Bourdeau and B.H.C. Cheng, ''A formal semantics for object model diagrams'', *IEEE Transactions on Software Engineering* **21** (10): 799–821 (1995).

[18] R.B. France and M.M. Larrondo-Petrie, ''From structured analysis to formal specifications: state of the theory'', in *Proceedings of 22nd ACM Annual Computer Science Conference* (*CSC '94*), ACM Press, New York (1994).

[19] K.C. Lano and H.P. Haughton, ''Integrating formal and structured methods in reverse engineering'', in *Proceedings of Working Conference on Reverse Engineering*, IEEE Computer Society, Los Alamitos, California, pp. 17–26 (1993).

[20] L. Semmens, T.W.G. Docker, and R.B. France, ''Integrating structured analysis and formal specification techniques'', Special Issue on Formal Methods, Part 2, *The Computer Journal* **35** (6) (1992).

[21] E. Yourdon, *Modern Structured Analysis*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1989).

[22] M. Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, New Jersey (1988).

[23] J.A. Goguen and R. Diaconescu, ''Towards an algebraic semantics of the object paradigm'', in *Proceedings of 9th International Workshop on Specification of Abstract Data Types*, H. Ehrig and F. Orejas (eds.), Lecture Notes in Computer Science, Vol. 785, Springer-Verlag, Berlin, pp. 1–29 (1994).

[24] J.A. Goguen, ''Sheaf semantics for concurrent interacting objects'', *Mathematical Structures in Computer Science* **2**: 159–191 (1992).

[25] P. Borba and J.A. Goguen, ''An operational semantics for FOOPS'', Technical Monograph PRG-115, Programming Research Group, Oxford University Computing Laboratory, Oxford (1994).

[26] M. Rahman, *Application of Object-Oriented Analysis and Design on Manufacturing System*, M.Sc. Project Report, Department of Computer Science, The University of Hong Kong, Hong Kong (1995).