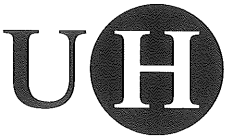


University of
Hertfordshire

CCS and Object-Oriented Concepts

M. Buchanan and R. G. Dickerson

July 1992

School of Information Sciences
Division of Computer Science
Technical Report No.140

CCS and Object-Oriented Concepts

Mary Buchanan and Bob Dickerson
Division of Computer Science, University of Hertfordshire
College Lane, Hatfield, Herts. AL10 9AB

July 1992

Abstract

The viability of using CCS as a formal specification language for classes of objects is investigated. The class based object-oriented paradigm is assumed throughout. It is concluded that CCS can be used to specify classes of objects and that it is particularly well suited for describing classes in which the time-ordering of operations is important. Further work is needed to evaluate the use of CCS to describe a complete system.

Sub-type inheritance can be expressed in CCS but at the expense of added complexity. Restriction inheritance can be expressed simply and clearly.

1 Introduction

CCS [Mil89] is a theory of communicating systems. "People will use it only if it enlightens their design and analysis of systems; therefore the experiment is to determine the extent to which it is *useful*, the extent to which the design process and analytical methods are indeed improved by the theory." [Mil89 page1]. This paper describes an experiment in which CCS is used to specify classes of objects and inheritance in the object-oriented paradigm.

2 Objects and Classes

2.1 Objects

An object has a set of operations and a state which remembers the effect of the operations; it also has a unique identity. Since the state of an object can be manipulated only by the operations exported via the object's interface, the details of the internal implementation of the state are hidden from the external view of the object. To define an object's interface it is necessary to consider the input events (the stimuli to which the object must react) and the output events (the responses the object should give).

2.2 Classes

In class-based object-oriented terminology, objects are considered to be instantiations of a class. A class can be considered to be a template describing the state and behaviour which all objects of the class have in common. A class also encompasses the concept of type. The set of operations declared in the class defines the interface of objects of the class; hence a class defines an abstract data type which can be used to determine the type compatibility of objects. Since many uniquely identifiable objects can be derived from a class, a class can also be considered to define a set of objects.

Although a class can be viewed as an implementation of an abstract data type, a class can be more than this. Whereas an abstract data type describes services provided by the type, a class can also describe services required from other classes. A class also tends to be a module and therefore to be separately compilable.

2.3 CCS and classes of objects

The existence of state within an object means that the order in which the object's operations are invoked can be important. Each object can therefore be regarded as an independent machine which can be described in terms of an equivalent state machine [Boo91 page 83]. Entity-life histories are useful to capture the time-ordering of actions in an informal manner [Buc90, Bri91, Dav90]. However, entity life histories have to be viewed in isolation since there is no way of composing components or of deriving the effects of such composition on the resultant system. We aimed to ascertain how useful CCS would be both to capture formally the time-ordering of operations on objects and to determine the effects of composing objects to form systems. In particular we wanted to establish whether CCS could be used to model inheritance.

In CCS the specification of state and the operations on that state are encapsulated together in agents which have their own identity. The behaviour of agents consists of discrete actions. A process is an independent agent which interacts with its environment solely by communication through its input and output ports. This seems analogous to the object-based concept [Weg88] in which an object has a set of operations and a remembered state which is accessed and amended by the operations alone. The *internal* behaviour of an object is hidden such that if two objects have the same external behaviour but different internal behaviour then, in an object-oriented system, either object can be substituted for the other without the behaviour of the system changing. (Here we are considering behaviour in terms of perceived functionality, not in terms of the time taken to achieve that functionality.)

A CCS agent can be expressed at different levels of abstraction; either in terms of its interaction with the environment or in terms of its composition from other agents. This is useful for system development since we can express the desired behaviour of a system as a single agent and also as the combination of simpler agents.

In CCS, the concept of *observation equivalence* expresses the equivalence of two processes which as stand-alone processes can perform the same pattern of external communications, but whose internal behaviour may differ. However, the observation equivalence of two processes, P and Q, is not enough to ensure that Q may be substituted for P in a system in which P is a component. The reason for this is that observation equivalence is not preserved by summation [Mil89 page 112]; summation is used in CCS to specify a choice of actions. Consider the expression

$$\tau.a \approx a$$

CCS describes the time-ordering of events but there is no sense of elapsed time. As a result, the fact that we might have to wait for the silent action τ to occur before we can perform the action a is of no significance. The expression τ.a means that we can perform the action a and so it is observation equivalent to the action a alone. However, consider the expression

$$a + b$$

Here, we can always perform the action a or the action b. From the above, $\tau.a \approx a$ but we cannot substitute $\tau.a$ for a in $a + b$.

$$\tau.a + b \not\approx a + b$$

If a τ action occurs, then the next action to occur must be an a. In other words we no longer have the choice between actions a or b because the τ action has precluded the b action.

For substitution to be possible we need congruence. Congruence is achieved if we have both observation equivalence and *stability*. A process P is stable if neither P itself nor any derivative of P has a leading τ action. Without stability we have the possibility that a τ action may cause a desired choice of actions to be overidden internally by the system such that choice is denied and one action is enforced. Pre-emptive 'τ's have proved troublesome in the work we describe on inheritance.

3 The Specification of Classes

We consider the viability of using CCS to specify classes of objects which have been specified informally using entity life histories. We have adopted the convention, used in Jackson System Development [Jac83], that the state of an entity can be read at any time and that this is assumed rather than specified explicitly.

3.1 Identity Cards

We shall consider a system in which a class Card is required. An object of class Card is used for identification purposes and when created it will have a card number. A card can be allocated, that is an identification for the owner of the card will be added to the card, and deallocated when the identification will be removed. At the end of its life, a card is deleted from the system. Figure 1. shows an entity life history depicting the class Card [Bri91].

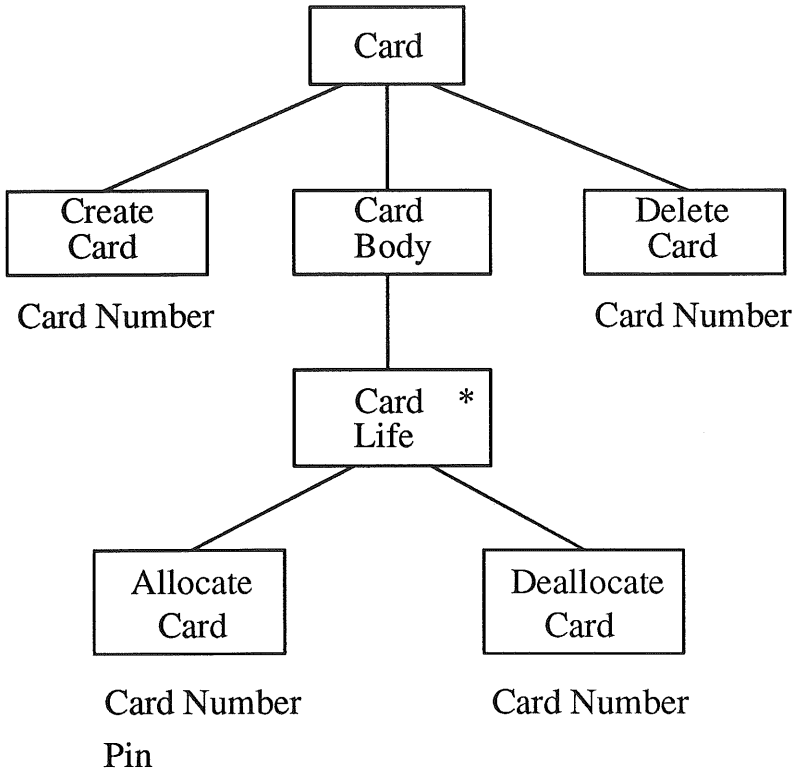


Figure 1: The Card Entity Life History

The Card class can be described in CCS as:

$$Card \stackrel{def}{=} createCard.Card1$$
$$Card1 \stackrel{def}{=} deleteCard.0 + allocateCard.deallocateCard.Card1$$

The CCS specification of the class Card is obviously concise and more importantly, it enforces the time-ordering of actions which is necessary for the particular class Card. Once a particular card has been created, it is in a new state ($Card1$) and in this state it cannot be created again. In state $Card1$, a card can perform the action *deleteCard* whereupon it reaches the state **0** from which it is incapable of further action, that is the card is no longer in the system. Alternatively, a card in state $Card1$ can perform the action *allocateCard* after which the only possible action is *deallocateCard* at which point the card is back in state $Card1$.

The CCS specification of card describes the operations or actions explicitly. However, the attributes or state variables of the card are implicit. The *createCard* action will result in a card having a card number. When *allocateCard* happens, the card attributes will be card number and some form of personal identity number (a PIN) and/or name, department or whatever is deemed appropriate for the system in hand. Since the attributes are implied, there is nothing to show what the attributes are. It is envisaged that such an abstract view of the data might cause problems in a specification aimed at object-oriented design.

We can add state variables (attributes) to the CCS specification by using the value passing features of the calculus:

$$Card \stackrel{def}{=} createCard(cn).Card1(cn)$$
$$Card1(cn) \stackrel{def}{=} deleteCard(cn).0 + allocateCard(cn, pn).deallocateCard(cn, pn).Card1(cn)$$

where the state variables are:

- cn a card number
- pn a personal identification number.

$Card$ is an agent in a state in which communication can occur at the port $createCard$ such that a value (from the set CardNumber) is received and becomes the value of the variable cn. Similarly, in state $Card1(cn)$ a value (from the set PIN) can be received at the port $allocateCard$ and becomes the value of the variable pn.

The variables in the equations are given scope in two ways. In the first, the variable in an input prefix, such as '$createCard(cn).$' , has scope in the agent expression which begins with the prefix. In the second way, the variable on the left-hand side of a defining equation has scope over the whole equation. For example, the variable cn has scope over the whole equation

$$Card1(cn) \stackrel{def}{=} deleteCard(cn).0 + allocateCard(cn, pn).deallocateCard(cn, pn).Card1(cn)$$

whereas the the variable pn is in an input prefix and has scope over the agent expression

$$allocateCard(cn, pn).deallocateCard(cn, pn).Card1(cn)$$

It should be emphasised that the value passing calculus is only an abbreviation for having a complete set of equations for each discrete card in the system:

$$\begin{aligned} Card &\stackrel{def}{=} \textstyle\sum_{c \in C} \sum_{p \in P} createCard_c.Card1_c \\ Card1_c &\stackrel{def}{=} deleteCard_c.0 + allocateCard_{c,p}.deallocateCard_{c,p}.Card1_c \end{aligned}$$

where

- C is the set of all values of type CardNumber and c is a member of C
- P is the set of all values of type Pin and p is a member of P.

$Card$ can thus be interpreted as the collection (or class) of all the individual card objects in the system.

We have tended not to use the value passing calculus since the concurrency work bench [Cle89] cannot be used to test transitions having parameters and also because the existence of the parameters increases the complexity of the equations.

3.2 Summary

To summarise, the CCS specification can be considered to describe the class $Card$. The operation $createCard$ would be requested from the class to create a card. The other actions describe the behaviour which all objects of the class share. The required time-ordering of actions is enforced.

4 Inheritance

Inheritance is one of the distinguishing features of the object-oriented paradigm. The different interpretations given to the meaning of inheritance depend largely on whether inheritance is being used to achieve sub-type hierarchies or to effect code-sharing without a type relationship.

4.1 Inheritance and the sub-type relationship

A type, the sub-type or descendant, can be derived from another type, the super-type or ancestor, such that the sub-type is a more specialised form of the super-type. In strict inheritance the sub-type will inherit all the attributes and operations of the super-type and will have additional attributes and/or operations. For example, the super-type *Card* could be inherited by a sub-type *CardPlus* containing the additional operation *changePin* to enable the pin number to be changed. Since *CardPlus* has all the properties of the parent class *Card*, it is a sub-type of *Card* and an instance of *CardPlus* can be used wherever an instance of *Card* is expected. The relationship between the instances of the types is referred to as an *is-a* relationship, we can say that an instance of type *CardPlus* is-a instance of type *Card*. The is-a relationship is transitive, hence if another type *CardPlus+* was created by inheritance from *CardPlus*, values of the type *CardPlus+* would stand in an is-a relationship to *Card* as well as to *CardPlus*. The is-a relationship is not symmetric and we cannot say that a *Card* is-a *CardPlus*.

We would want users (clients) of the sub-type inheritance hierarchy to be aware of the inheritance structure and the relationships between the ancestors and descendants. A descendant is always a more specialised form of a more general ancestor.

The interface of a class defines the specification for the class such that the profiles of the exported operations are declared. A class conforms to another if it can be used in all contexts where the other class is expected. Conformance depends only on the interfaces of the classes and not on the implementation; the interface of a class subsumes the interface of any class to which it conforms. For sub-type inheritance, it is essential that the sub-class inherits the interface of the super-class whereas the implementation may or may not be inherited. In languages such as Trellis/Owl and Emerald [Atk91 page 15] conformance is used as the sole basis for defining type compatibility and such languages could therefore support sub-type inheritance of interfaces only. However, in Eiffel classes must have implementations as well as interfaces related by inheritance if the classes are to be type-compatible.

4.2 Inheritance and the like relationship

4.2.1 Restriction

In this interpretation of inheritance, simpler less specialised classes can be created from more complex specialised classes. If, for example, the *CardPlus* class existed and there was a requirement to create a *Card* class, then this could be achieved by inheriting the code from the *CardPlus* class and restricting the *changePin* operation. We have reversed the inheritance structure from that described in the previous section on sub-typing since now we have that *CardPlus* is the super-class and *Card* is the sub-class. The relationship between the super-class and sub-class can be viewed as a *like* relationship. In the example given, we could say that a *Card* is like a *CardPlus* but we would not be able to use an instance of the sub-class *Card* wherever an instance of the class *CardPlus* was expected. However, the super-class *CardPlus* is a sub-type of the sub-class *Card* and languages such as Emerald and Trellis/Owl would presumably allow the assignment of an instance of class *CardPlus* to an instance of class *Card* since *CardPlus* conforms to *Card*.

If the like relationship is to hold when using inheritance and restriction, we again have that the interface must be inherited but that the implementation need not be.

4.2.2 Restriction and Enrichment

If a new class is derived via inheritance such that not only are operations in the super-class restricted but also new operations are added to the sub-class, then the sub-class may still have a *like* relationship to the super-class but the sub-class will no longer be a simpler, less specialised

version of the super-class. Consequently, the super-class will no longer be a sub-type of the sub-class. For example, if a class *AnotherCard* inherits from class *CardPlus* such that the operation *changePin* is restricted and another operation, such as *addAddress*, is added to the new class, then *AnotherCard* is still like *CardPlus* but *CardPlus* is not a sub-type of *AnotherCard*.

To maintain a like relationship, we again require that the interface is inherited whether or not the implementation is inherited.

4.3 Inheritance without a semantic relationship

It is possible to use inheritance for the sole purpose of reusing the code of an existing class in a new class; there need not be a semantic relationship between the classes. If, for example, a class Dequeue existed in which a double ended queue was defined, then this could be inherited by a class, Stack, to define a stack. All of the Dequeue code would be inherited by the class Stack, but the operation to enable an item to join the back of the queue would be restricted in order to prevent an item from being added to the bottom of a stack. The dequeue operations would probably be renamed to those more commonly used for stacks, for example *head* would be renamed *top*. A stack is not like a dequeue and there is no type compatibility between the classes; the inheritance relationship should therfore be hidden from clients.

For this type of inheritance, it is essential that the implementation code is inherited but it is unlikely that the interface will be inherited.

4.4 Inheritance and specification

As regards specification we can view the potential benefits of inheritance from two perspectives. Conceptually, it will be easier to understand a specification in which sub-type inheritance is used as an abstraction to describe specialisation-generalisation relationships which exist between classes. Once the behaviour of a base class has been understood, it is necessary only to understand the new behaviour added. "Inheritance may be applied to explicitly express commonality, beginning with the early activities of analysis" [Coa91]. Pragmatically, if extra capability is required for the same or for a new class after a specification has been written, it is attractive to have to describe only the new behaviour as an extension of the old and not to have to completely respecify.

Similarly, if there is a need to create a simpler version of a class (such as a *Card* from a *CardPlus*) where there is a semantic relationship between the classes, then it could be an advantage to use inheritance because it is only necessary to understand what behaviour has been removed.

However, it would seem undesirable to use inheritance in specifications merely to reuse code without any semantic relationship between the classes. Great care would be needed in order to convey the intended change of semantics between the classes and it would seem likely that greater clarity could be obtained by other means.

5 CCS and sub-type inheritance

In this paper we consider inheritance mainly from the sub-type point of view such that *CardPlus* is a sub-type of the more general class *Card*. We investigate the extent to which CCS can model such inheritance.

5.1 A new class defined to extend the behaviour of class Card

Suppose we want to define another class, *CardPlus* having the same behaviour as *Card* but with the additional operation that the PIN can be changed. An entity life history for *CardPlus* is shown in Figure 2.

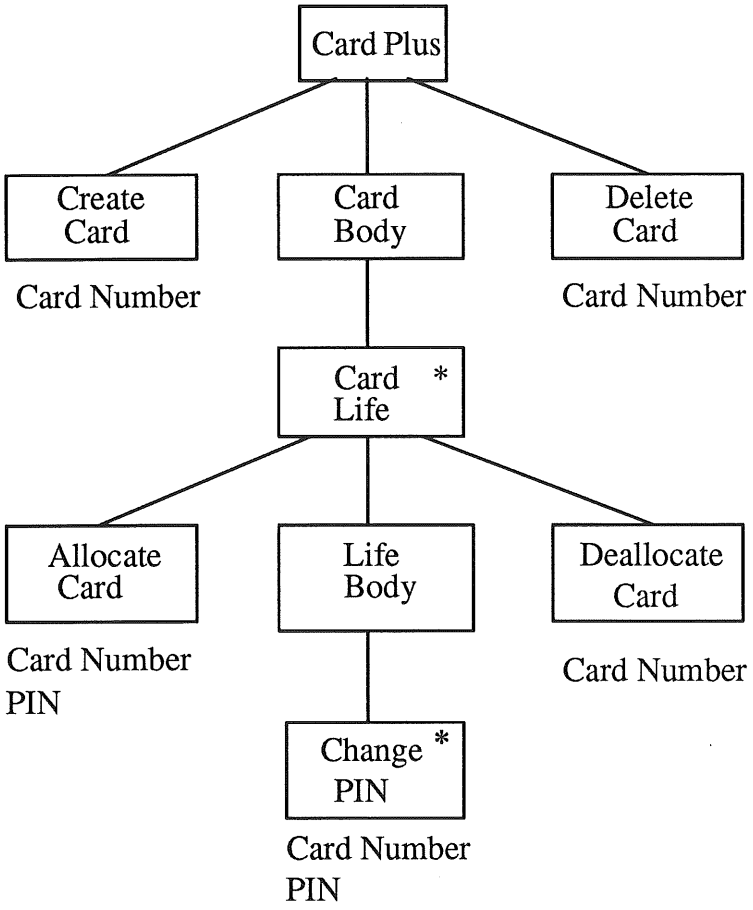


Figure 2: The CardPlus Entity Life History

The CCS specification for *CardPlus* is:

$$
\begin{aligned}
CardPlus &\stackrel{def}{=} createCard.CardPlus1 \\
CardPlus1 &\stackrel{def}{=} deleteCard.0 + allocateCard.CardPlus2 \\
CardPlus2 &\stackrel{def}{=} changePin.CardPlus2 + deallocateCard.CardPlus1
\end{aligned}
$$

From this specification, we can see that the *changePin* action can only be applied after a card has been allocated and before it is deallocated but that within these constraints, the PIN can be changed as often as wanted or it need never be changed.

In order to simplify the following discussion, we omit the *createCard* and *deleteCard* actions. The adapted *Card* class, *CardA*, is defined as:

$$CardA \stackrel{def}{=} allocateCard.deallocateCard.CardA$$

and similarly, the adapted CardPlus class is defined as:

$$CardPlusA \stackrel{def}{=} allocateCard.CardPlusA1$$
$$CardPlusA1 \stackrel{def}{=} changePin.CardPlusA1 + deallocateCard.CardPlusA$$

5.2 Inheriting the behaviour of class Card

We aim to establish the viability of using CCS to represent inheritance. We want to ascertain whether it is possible to form a new class, $CardH$, which inherits behaviour from $CardA$ and has the added behaviour that the PIN can be changed. $CardH$ is to be a sub-type of the super-type $ClassA$ and is to be congruent with $CardPlusA$ so that in a given specification the two would be interchangeable.

The only way that agents can communicate with one another is via their ports. Accordingly, we define a new agent which is congruent with $CardA$ but which has the potential for communication through ports other than $allocateCard$ and $deallocateCard$.

Consider the agents:

$$CardC \stackrel{def}{=} allocateCard.\overline{a}.b.CardC$$
$$CardD \stackrel{def}{=} a.deallocateCard.\overline{b}.CardD$$

If we compose $CardC$ and $CardD$, and restrict the ports a and b so that only internal communication is possible via these ports, we have

$$CardCD \stackrel{def}{=} (CardC|CardD)\backslash\{a,b\}$$

Figure 3 shows $CardCD$ in diagrammatic form.

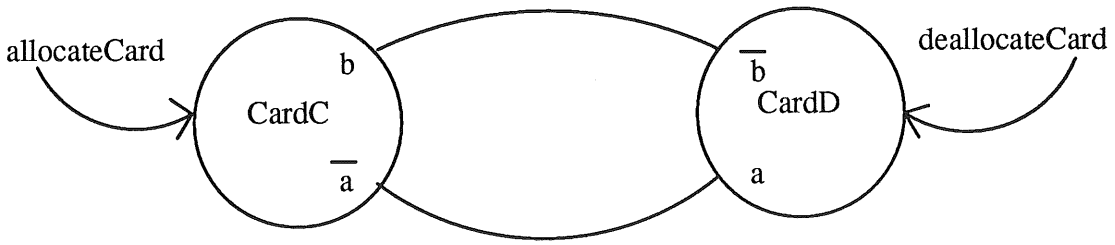


Figure 3: The CardCD

We can use the expansion theorem [Mil89 page 69] to examine the behaviour of the concurrent (i.e. interleaved) processes,

$$\begin{aligned}
CardCD &= (CardC|CardD)\backslash\{a,b\} \\
&= (allocateCard.\overline{a}.b.CardC|a.deallocateCard.\overline{b}.CardD)\backslash\{a,b\} \\
&= (allocateCard.(\overline{a}.b.CardC|a.deallocateCard.\overline{b}.CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.(b.CardC|deallocateCard.\overline{b}.CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.deallocateCard.(b.CardC|\overline{b}.CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.deallocateCard.\tau.(CardC|CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.deallocateCard.\tau.CardCD)\backslash\{a,b\}
\end{aligned}$$

By the τ law $a.\tau.P = a.P$ (which holds for observation equivalence), the silent action τ can be be ignored when it occurs between actions, hence

$$CardCD \quad = allocateCard.deallocateCard.CardCD$$

and since

$$CardA \quad = allocateCard.deallocateCard.CardA$$

we can observe that $CardA$ and $CardCD$ are equal and so must be congruent. The significance of this is that if in a system we have $CardA$, we can substitute $CardCD$ for $CardA$. We shall use $CardCD$ to try and establish if we can use it in an inheritance relationship. We want to define a new agent having the *changePin* behaviour in such a way that we can compose it with $CardCD$ such that we have congruence with $CardPlusA$.

Consider $CardE$ defined as:

$$\begin{aligned} CardE &\stackrel{def}{=} a.CardE1 \\ CardE1 &\stackrel{def}{=} changePin.CardE1 + \overline{c}.CardE \end{aligned}$$

Now in order to link $CardE$ in to $CardCD$, we relabel the a port in $CardD$:

$$CardDa \stackrel{def}{=} CardD[c/a]$$

Finally, we define $CardH$ as:

$$CardH \stackrel{def}{=} (CardC|CardDa|CardE)\backslash\{a, b, c\}$$

Figure 4 shows $CardH$ in diagrammatic form.

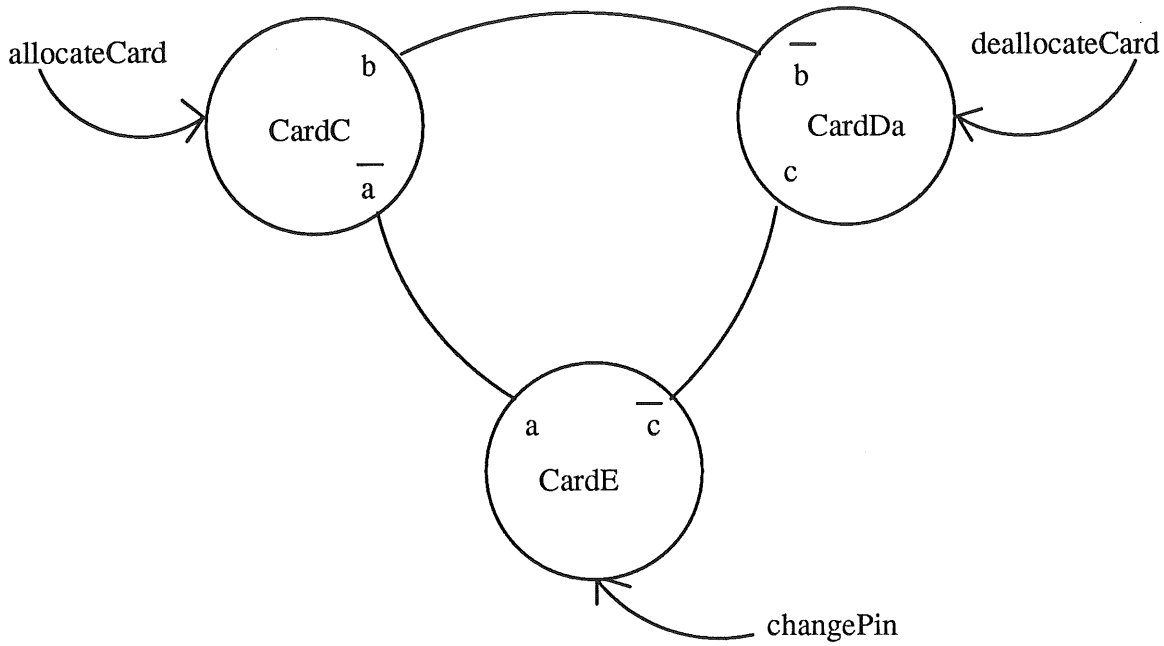


Figure 4: The CardH

However, *CardH* does not exhibit the desired behaviour. Testing on the concurrency workbench [Cle89] shows that *CardH* and *CardPlusA* are not observation equivalent. The concurrency workbench can also test for preorder relationships as defined in [Hen88]. Thus we have that

$$CardH \approx_{may} CardPlusA$$

which means that *CardH* and *CardPlusA* *may* (not must) be equivalent.

Although the traces of the alternative sequences of actions which can be undertaken by *CardH* and *CardPlusA* are the same, this only tells us that it is possible for *CardH* and *CardPlusA* to follow the same sequences of actions, not that they must be able to follow such sequences at all times.

We also have that

$$CardH \sqsubseteq_{must} CardPlusA$$

This tells us that there are no actions which *CardH* can perform that *CardPlusA* cannot also perform but tells us nothing about the extra actions that *CardPlusA* can perform over and above tthe actions of *CardH*. For *CardH* to be able to simulate *CardPlusA* , we would need to have observation equivalence or failing that we would want to have that the behaviour of *CardH* was a super-set of the behaviour of *CardPlusA*, that is $CardPlusA \sqsubseteq_{must} CardH$; however testing shows that this is false.

When we consider the expansion theorem, we discover that *CardPlusA* must allow one to *allocateCard*, *changePin* and *deallocateCard* whereas *CardH* must allow one to *allocateCard* and *deallocateCard* but that it may or may not allow one to *changePin*.

Thus by the expansion theorem,

$$
\begin{aligned}
CardH &\stackrel{def}{=} (CardC|CardDa|CardE)\backslash\{a,b,c\}\\
CardH &= (allocateCard.\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.CardE1)\backslash\{a,b,c\}\\
CardH &= (allocateCard.(\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.cardE1))\backslash\{a,b,c\}\\
CardH &= (allocateCard.\tau.(b.CardC|c.deallocateCard.\overline{b}.CardD|\\
&\qquad (changePin.CardE1 + \overline{c}.CardE)))\backslash\{a,b,c\}\\
CardH &= (allocateCard.\tau.(\tau.(b.CardC|deallocateCard.\overline{b}.CardD|CardE)\\
&\qquad + \\
&\qquad changePin(b.CardC|c.deallocateCard.\overline{b}.CardD|CardE1)))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

Now we can see where the problem lies. The silent action after the *allocateCard* action is of no concern but it is followed by a choice of actions, the first of which has a leading τ. If the leading τ occurs , then we no longer have a choice of action between the *changePin* and the *deallocateCard* actions since the *changePin* action will have been pre-empted. Taking the expansion one stage further makes this clearer.

$$
\begin{aligned}
CardH &= (allocateCard.\tau.(\tau.deallocateCard(b.CardC|.\overline{b}.CardD|CardE)\\
&\qquad + \\
&\qquad changePin.(\tau.(b.CardC|deallocateCard.\overline{b}.CardD|a.CardE1)\\
&\qquad\qquad + \\
&\qquad\qquad changePin(b.CardC|c.deallocateCard.\overline{b}.CardD|\\
&\qquad\qquad\qquad\qquad a.CardE1))))\backslash\{a,b,c\}
\end{aligned}
$$

Once the leading τ has occurred, via an autonomous action of the system, then the next action which can be taken from outside the system must be a *deallocateCard* regardless of whether this is the action actually wanted.

5.3 The problem with CardH

The problem of the silent τ action is caused by the agent $CardE$ defined as:

$$
\begin{aligned}
CardE &\stackrel{def}{=} a.CardE1\\
CardE1 &\stackrel{def}{=} changePin.CardE1 + \overline{c}.CardE
\end{aligned}
$$

The presence of $\overline{c}.CardE$ in $CardE1$ lies at the heart of the matter. The $\overline{c}$ means that silent and secret communication can occur with $CardDa$. Such an action is beyond the control of the environment in which $CardH$ finds itself so the behaviour of $CardH$ is unpredictable. If the silent communication with $CardDa$ occurs then the *changePin* action is denied to the environment, without the environment being aware that this is so.

5.4 A Partial Solution

Various attempts were made to overcome the problem of the unwanted pre-emptive τ but without success. However, we were able to arrive at a partial solution; the solution is partial because it leads to re-specification of the original system. In order to prevent the unwanted silent communication between $CardE$ and $CardDa$, we introduce an external action such that the environment has to actively request that it wants to deallocate a card *before* actually activating the *deallocateCard* action. In order to do this, $CardE$ is replaced by $CardF$ defined as:

$$CardF \stackrel{def}{=} a.CardF1$$
$$CardF1 \stackrel{def}{=} changePin.CardF1 + requestDeallocate.\overline{c}.CardF$$

The effect of the *requestDeallocate* action is to prevent the unwanted silent communication via the $\overline{c}$. After an external *requestDeallocate* action has been chosen, a wanted silent communication is enabled and has the desired effect that the next permitted action is *deallocateCard*. Consider *CardI* as shown diagrammatically in Figure 5.

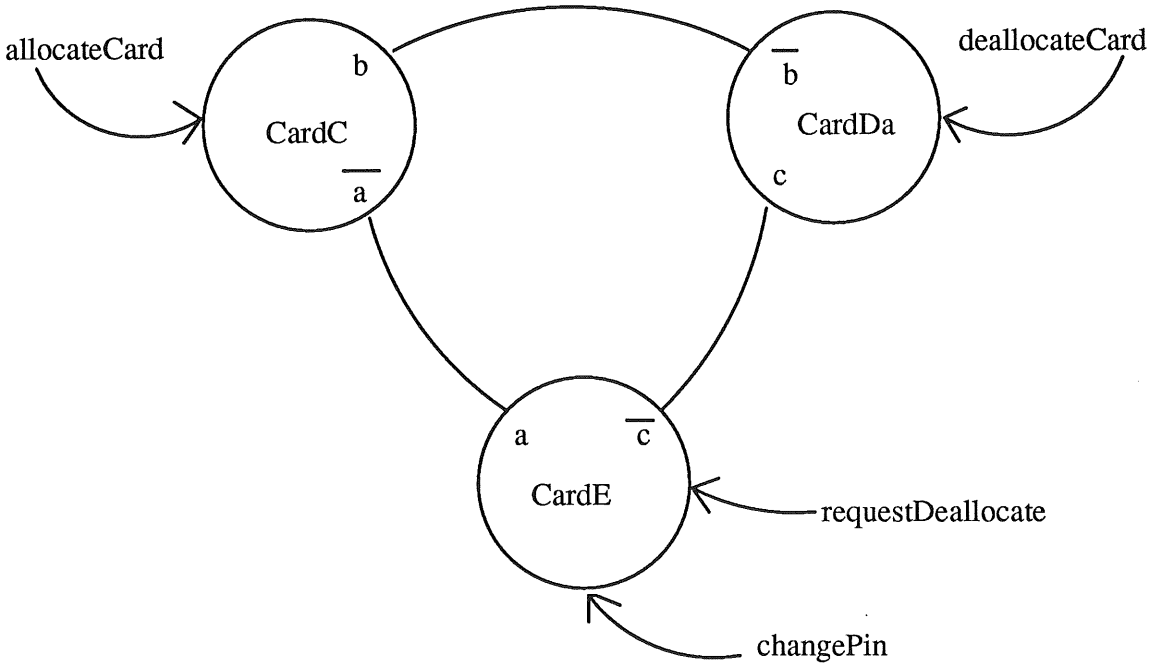


Figure 5: The CardI

CardI is defined in CCS as:

$$CardI \stackrel{def}{=} (CardC|CardDa|CardF)\backslash\{a,b,c\}$$

By the expansion theorem,

$$\begin{aligned}
CardI &= (allocateCard.\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.cardF1)\backslash\{a,b,c\}\\
CardI &= (allocateCard.(\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.cardF1))\backslash\{a,b,c\}\\
CardI &= (allocateCard.\tau.(b.CardC|c.deallocateCard.\overline{b}.CardD|\\
&\qquad (changePin.CardF1 + requestDeallocate.\overline{c}.CardF)))\backslash\{a,b,c\}\\
CardI &= (allocateCard.\tau.(changePin.(b.CardC|c.deallocateCard.\overline{b}.CardD|CardF1)\\
&\qquad +\\
&\qquad requestDeallocate.(b.CardC|c.deallocateCard.\overline{b}.CardD|\overline{c}.CardF)))\\
&\qquad\qquad \backslash\{a,b,c\}
\end{aligned}$$

We can now see that it is only after a *requestDeallocate* that *CardDa* and *CardF* can communicate via a silent action as shown below.

$$
\begin{aligned}
CardI \;=\; & (allocateCard.\tau. \\
& \quad (changePin. \\
& \qquad (changePin.(b.CardC|c.deallocateCard.\overline{b}.CardD|CardF1) \\
& \qquad + \\
& \qquad requestDeallocate.(b.CardC|c.deallocateCard.\overline{b}.Card| \\
& \qquad\qquad \overline{c}.CardF)) \\
& \quad + \\
& \quad requestDeallocate.\tau.(b.CardC|deallocateCard.\overline{b}.CardD|CardF))) \\
& \qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

$$
\begin{aligned}
CardI \;=\; & (allocateCard.\tau. \\
& \quad (changePin. \\
& \qquad (changePin. \\
& \qquad\quad (changePin(b.CardC|c.deallocateCard.\overline{b}.CardD| \\
& \qquad\qquad CardF1) \\
& \qquad\quad + \\
& \qquad\quad requestDeallocate.(b.CardC|c.deallocateCard.\overline{b}. \\
& \qquad\qquad CardD|\overline{c}.CardF)) \\
& \qquad + \\
& \qquad requestDeallocate.\tau.(b.CardC|deallocateCard.\overline{b}.CardD| \\
& \qquad\qquad CardF)) \\
& \quad + \\
& \quad requestDeallocate.\tau.deallocateCard.(b.CardC|\overline{b}.CardD|a.CardF1))) \\
& \qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

$$
\begin{aligned}
CardI \;=\; & (allocateCard.\tau. \\
& \quad (changePin. \\
& \qquad (changePin. \\
& \qquad\quad (changePin. \\
& \qquad\qquad (changePin(b.CardC|c.deallocateCard.\overline{b}. \\
& \qquad\qquad\quad CardD|CardF1) \\
& \qquad\qquad + \\
& \qquad\qquad requestDeallocate.(b.CardC| \\
& \qquad\qquad\quad c.deallocateCard.\overline{b}.CardD|\overline{c}.CardF)) \\
& \qquad\quad + \\
& \qquad\quad requestDeallocate.\tau.(b.CardC|deallocateCard.\overline{b}. \\
& \qquad\qquad CardD|CardF)) \\
& \qquad + \\
& \qquad requestDeallocate.\tau.deallocateCard.(b.CardC|\overline{b}.CardD| \\
& \qquad\qquad a.CardF1)) \\
& \quad + \\
& \quad requestDeallocate.\tau.deallocateCard.\tau.(CardC|CardD|CardF))) \\
& \qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

By defining $CardI$, we have succeeded in removing the instability in $CardH$ caused by the leading τ. However, $CardI$ does not match the specification of $CardPlusA$ due to the presence of the extra action, $requestDeallocate$. $CardI$ is in fact congruent with $CardPlusF$ defined as:

$$
\begin{aligned}
CardPlusF &\stackrel{def}{=} allocateCard.CardplusF1 \\
CardPlusF1 &\stackrel{def}{=} changePin.CardPlusF1 + requestDeallocate.deallocateCard.CardPlusF
\end{aligned}
$$

We can regard $CardPlusF$ as a more friendly version of $CardPlusA$ in that the $requestDeallocate$

could be interpreted as giving a user a warning message such as "Are you sure you want to deallocate this card? Such an action will result in the card having to be reallocated".

Tests on the concurrency workbench confirm that $CardI$ is congruent with $CardPlusF$. $CardI$ can be considered to have inherited the behaviour of $CardCD$ and to have extended the behaviour to include the *changePin* and *requestDeallocate* operations. If in a system we had $CardA$ and wanted to inherit from it to form a class having the behaviour of $CardPlusF$, then we could replace $CardA$ with $CardCD$ (since they are congruent) and use $CardCD$ as the supertype from which to derive the subtype $CardI$. Since $CardI$ is congruent with $CardPlusF$, $CardI$ has the behaviour required of the new class.

5.5 A Single ChangePin Action

It should be stressed that the problems caused in $CardH$ by the unwanted pre-emptive τ arose because we wanted to be able to change the PIN more than once. No such problems arise if one specifies that the system must change the PIN once and once only. Consider such a card defined as:

$$SimpleCardPlus \stackrel{def}{=} allocateCard.changePin.deallocateCard.SimpleCardPlus$$

We can define $CardG$ as:

$$CardG \stackrel{def}{=} a.changePin.\overline{c}.CardG$$

Then $CardCDG$ is:

$$CardCDG \stackrel{def}{=} (CardC|CardDa|CardG)\backslash\{a,b,c\}$$

By the expansion theorem,

$$\begin{aligned}
CardCDG &= (allocateCard.\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.changePin.\overline{c}CardG) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.(b.CardC|c.deallocateCard.\overline{b}.CardD|changePin.\overline{c}.CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.(b.CardC|c.deallocateCard.\overline{b}.CardD|\overline{c}.CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.\tau.(b.CardC|deallocateCard.\overline{b}.CardD|CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.\tau.deallocateCard.(b.CardC|\overline{b}.CardD|CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.\tau.deallocateCard.\tau.(CardC|CardD|CardG)) \\
&\qquad \backslash\{a,b,c\}
\end{aligned}$$

Since all the τ are non-leading, they can be disregarded and $CardCDG$ is seen to be congruent with $SimpleCardPlus$.

6 Interrupts

Unfortunately, CCS has no way to model interrupts. Using the notation P^iQ to represent P interrupted by Q, we have that P^iQ behaves like P until Q does anything at all whereupon it behaves like Q. An interrupt operator might have enabled us to solve the problem in $CardH$

by allowing the *deallocateCard* operation to interrupt $CardH$ such that $CardH$ returned to its original state. Consider $CardH_2$ defined as:

$$CardH_2 \stackrel{def}{=} ((CardC|CardE_2)\backslash\{a,b\})^i deallocateCard.CardH_2$$

where

$$\begin{aligned} CardE_2 &\stackrel{def}{=} a.CardE_22 \\ CardE_22 &\stackrel{def}{=} changePin.CardE_22 \end{aligned}$$

$CardE_2$ is the same as $CardE$ except that the c link with $CardD$ has been removed. The behaviour of $CardH_2$ will be that of $(CardC|CardE_2)\backslash\{a,b\}$ until a *deallocateCard* action occurs and the original state is resumed. However, the specification is still not quite right since a *deallocateCard* can occur before an *allocateCard*. To overcome this, we define $CardH_3$ as:

$$CardH_3 \stackrel{def}{=} allocateCard.\left(((\overline{a}.b.CardC|CardE_2)\backslash\{a,b\})^i deallocateCard.CardH_3\right)$$

Although such a specification would give us the behaviour we want, we are perhaps moving away from inheritance in that we are only using $CardC$ and not $CardD$. Instead of inheriting all the behaviour of $CardCD$, we are only inheriting the behaviour of $CardC$.

7 The π-Calculus

The π-Calculus [Mil91] enables communication between agents to carry information which changes the linkage between the agents and thereby can describe agents which have a changing structure. This would seem to offer a means of expressing inheritance.

The major advance over CCS is the ability to send the names of links as parameters in communications. A link is formed between agents having complementary labels to ports. No distinction is made between link names, variables and ordinary data values; they are all just *names*. Thus there are only two essential classes of entity: names and agents.

"It is considered that the π-calculus will lead to a better understanding of object-oriented programming " [Mil91 Page2].

7.1 A First Attempt

We want to define our basic $Card$ so that it can receive new actions. Consider $CardB$ defined as:

$$\begin{aligned} CardB &\stackrel{def}{=} allocateCard.CardB1 \\ CardB1 &\stackrel{def}{=} a(x).x.CardB1 + deallocateCard.CardB \end{aligned}$$

$a(x).P$ means that agent P can receive a name z (of a link or of a value) at port a and then behaves as $P\{z/x\}$, that is all parameters x in P are replaced by the actual name z. Thus a new link x can be sent to $CardB$ along the link a, then the new link can be used. In the case where we want a *changePin* action sent to $CardB$, we can define:

$$CardS \stackrel{def}{=} \overline{a}changePin.CardS$$

$\overline{a}t$ is interpreted to mean transmit the value t along the link a.

We then compose as:

$$CardBS \stackrel{def}{=} (a)(CardB|CardS)$$

Restriction in the π-calculus is expressed as $(a)P$ meaning that external actions at the ports a and $\overline{a}$ are prohibited but internal communications along a are permitted for the components of P.

Unfortunately, when we apply the expansion theorem to $CardBS$, we discover that a leading τ occurs such that the *deallocateCard* action can be pre-empted.

In order to overcome the pre-emptive τ, we define $CardR$ such that there is a guard (*newAction*) on the $a(x)$ as:

$$\begin{aligned} CardR &\stackrel{def}{=} allocateCard.CardR1 \\ CardR1 &\stackrel{def}{=} newAction.a(x).x.CardR1 + deallocateCard.CardR \end{aligned}$$

(We considered putting the guard, *sendChangePin*, on the $\overline{a}$ in $CardS$ but this caused problems such as the possibility of a *sendChangePin* after a *deallocateCard*.)

$$CardS \stackrel{def}{=} \overline{a}changePin.CardS$$

By composition,

$$CardRS \stackrel{def}{=} (a)(CardR|CardS)$$

By expansion,

$$\begin{aligned} CardRS &= (a)(allocateCard.CardR1|\overline{a}changePin.CardS) \\ CardRS &= allocateCard.(a)(CardR1|\overline{a}changePin.CardS) \\ CardRS &= allocateCard.(a)((newAction.a(x).x.CardR1 \\ &\qquad + \\ &\qquad deallocateCard.CardR)|\overline{a}changePin.CardS) \\ CardRS &= allocateCard.(a)((newAction.a(x).x.CardR1|\overline{a}changePin.CardS) \\ &\qquad + \\ &\qquad deallocateCard.CardR|\overline{a}changePin.CardS) \\ CardRS &= allocateCard.(a)((newAction.(a(x).x.CardR1|\overline{a}changePin.CardS) \\ &\qquad + \\ &\qquad deallocateCard.(allocateCard.CardR1|\overline{a}changePin.CardS)) \\ CardRS &= allocateCard.(a)(newAction.\tau.(changePin.CardR1|CardS) \\ &\qquad + \\ &\qquad deallocateCard.allocateCard.(CardR1|\overline{a}changePin.CardS)) \\ CardRS &= allocateCard.(a)(newAction.\tau.changePin.(CardR1|CardS) \\ &\qquad + \\ &\qquad deallocateCard.allocateCard.(CardR1|\overline{a}changePin.CardS)) \end{aligned}$$

We can see that we have the behaviour we require in that after an *allocateCard* action has occurred, we can now perform the *changePin* action as an alternative choice to a *deallocateCard* action. Furthermore, a *changePin* action can never be performed immediately after a card has been deallocated.

However, whenever we wish to perform a *changePin* action we have to go through the *newAction* action and send the *changePin* action to $CardR$. This seems unnecessarily laborious. What we really want is to send the *changePin* action to the $CardR$ once only and then use it repeatedly as required.

7.2 First refinement to CardRS

$$
\begin{aligned}
CardR_2 &\stackrel{def}{=} allocateCard.CardR_21 \\
CardR_21 &\stackrel{def}{=} newAction.a(x).CardR_22 + deallocateCard.CardR_2 \\
CardR_22 &\stackrel{def}{=} x.CardR_22 + deallocateCard.CardR_2
\end{aligned}
$$

$CardR_21$ enables the *deallocateCard* action to be chosen in which case an *allocateCard* action must be the next action chosen. Alternatively, *newAction* may be chosen in which case a new link x can be received. $CardR_22$ enables the x action to be performed repeatedly, without the need to use $newAction.a(x)$ each time, and also enables *deallocateCard* to be selected when required. We can compose $CardR_2$ with $CardS_2$ defined as:

$$CardS_2 \stackrel{def}{=} \overline{a}changePin.0$$

Now that we do not have to repeatedly send a *changePin* action to $CardR$, we have been able to simplify the $CardS$ definition to $CardS_2$.

By composition,

$$CardRS_2 \stackrel{def}{=} (a)(CardR_2|CardS_2)$$

7.3 Second refinement to CardRS

So far we have been discussing the case where we could add one new action to the *Card*. In order to have more flexibility, we need to be able to send more than one new action to the card $CardR$.

$$
\begin{aligned}
CardR_3 &\stackrel{def}{=} allocateCard.CardR_31 \\
CardR_31 &\stackrel{def}{=} newAction.a(x).(CardR_31\{x'/x\} + CardR_32) \\
&\quad + \\
&\quad deallocateCard.CardR_3 \\
CardR_32 &\stackrel{def}{=} x.CardR_32 + CardR_31\{x'/x\}
\end{aligned}
$$

Now it would appear that we can send two new actions to $CardR_3$ via $a(x)$ and $a(x')$. However, it is not so clear what happens with $CardR_32$; will the last action received (x') override the first action? We have not yet resolved this question and it may well be that the π-calculus can provide more elegant solutions. There is the added complexity that when new actions are added, the order in which the actions are used could well be important.

8 Data addition

Until now we have not needed to add new data to the *Card*, only new actions. Thus *changePin* was added to enable the *PIN* which was already in the *Card* to be changed. However, there are times when we might want to add new data as well as a new process. Consider the requirement that the *Card* is to contain information about the rooms that the cardholder is entitled to enter in a secure building. We need to add data about rooms, since this is not on the original card, and an action to add rooms to the original set of rooms. We will also want to be able to delete rooms from the set. We have not yet tackled these issues, but it does appear that the π-calculus may be able to address such matters.

In CCS, if one adds an operation such that a new type of data can arrive at a port, then one has implied that a corresponding new attribute has also been added to the agent.

9 Discussion

9.1 Liveness and Safety

We can say that the liveness of a system describes the desirable behaviour it must have, whereas the safety of a system describes the undesirable behaviour it must not have. We have been trying to avoid having a pre-emptive τ because the liveness of $CardplusA$ was not preserved by $CardH$ due to such a τ but there are circumstances in which a pre-emptive τ might be desirable. It might be that one wants a system to make decisions about what it did without reference to the environment. In [Bai91] the specification of a level-crossing is considered. It is required to model the situation where an observer sees either a train or cars approaching the level crossing and is never prevented from seeing either, but cannot choose between them. It would be desirable if the choice could be made internally by the system such that when, for example, an approaching train is sensed, actions are performed which will eventually allow the train to cross.

In the case of $CardH$, it might be desirable for the system to decide that for some well-defined reason the user must not be allowed to change his pin number. However, with $CardH$ we have modelled the situation in which the decision to prevent the change pin action is taken by the system purely *at random*. This is not a desirable behaviour.

9.2 CCS and Inheritance

9.2.1 Sub-type inheritance

We have had only limited success at using CCS to define sub-type inheritance and even that was achieved at the expense of adding much extra complexity to the parent $Card$ class specification in the form of extra ports. In addition, we had to add the $requestDeallocate$ action to the original specification in order to achieve stability.

9.2.2 Inheritance and Restriction

The restriction operator in CCS does enable inheritance to be modelled very simply, provided the new class is a restricted version of the existing class and no extra behaviour is added. If the class $CardPlusA$ was in existence and there was a requirement to provide the simpler class $NewCardA$, having the behaviour of $CardA$, then this can be achieved as:

$$NewCardA \stackrel{def}{=} CardPlusA\backslash\{changePin\}$$

$NewCardA$ is observation equivalent to $CardA$.

9.3 CCS and State

In order to show what happens to the state variables as the result of an action, it is necessary to use the value passing calculus. This has the overhead that the specifications become considerably more cumbersome, particularly if many state variables are required. There is the added disadvantage that the concurrency work bench does not handle the value passing calculus.

10 Conclusions

If inheritance is to be used in specifications, then it should aim to reduce complexity both in the semantic relationships between classes and in the specification code itself. This is particularly important when one considers that classes defined at the specification stage might not necessarily be those implemented for the final system. We have only been able to specify sub-type inheritance in CCS by introducing extra complexity into the specification. For example, in order to inherit from *CardA* we had to replace it with the more complex *CardCD*. However, inheritance can be expressed naturally and simply in CCS in the limited case where the new class is simply a restriction of the super-class.

The fact that we have had difficulty in expressing sub-type inheritance in CCS should not overshadow the benefits which CCS can bring to an object-oriented specification. The time-ordering of operations can be enforced where required. In addition, the labels of a class give all the operations for objects of the class and likewise the labels of a system give all the operations of the system. Restrictions on the labels show the operations which are internal to the system. The composition operator in the calculus makes it possible to determine the effects that concurrent objects will have on one another; as we have discovered, such effects are not always obvious from the initial specifications of the objects. However, we are not yet sure whether the effects are a reflection on the system being developed or whether they have arisen from our CCS model of the system.

The π-calculus seems to offer a much simpler means than CCS of building adaptability into a specification, although this adaptability is perhaps modelling extendibility rather than inheritance.

11 References

[Atk91] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution.* Addison-Wesley 1991

[Bai91] Jean Baillie. *A CCS Case Study: A Safety Critical System.* Software Engineering Journal, vol 6, number 4, July 1991. IEE and BCS.

[Boo91] Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings 1991

[Bri91] Carol Britton and Mary Buchanan. *Modelling Techniques for Object-Oriented Design.* Technical Report No.133, Hatfield Polytechnic, 1991

[Buc90] Mary Buchanan. *The Phantom of the Object.* MSc Project Report, Hatfield Polytechnic, 1990

[Cle89] Cleaveland, Parrow and Steffen. *The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems.* Technical Report ECS-LFCS-89-83, LFCS, Department of Computer Science, University of Edinburgh, August 1989.

[Coa91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice-Hall International 2nd edition 1991

[Dav90] N.W.Davis, M.Irving and J.E.Lee. *The evolution of object-oriented design from concept to method.* In *Managing Complexity in Software Engineering*, ed R.J.Mitchell, Peter Peregrinns Ltd on behalf of the Institution of Electrical Engineers, 1990

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988

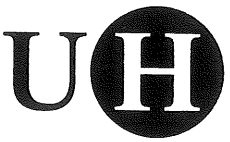
[Jac83] Michael Jackson. *System Development.* Prentice Hall International, 1983

[Mil89] R.Milner. *Communication and Concurrency.* Prentice Hall International, 1989

[Mil91] Robin Milner. *The Polyadic π-Calculus: A Tutorial.* Report ECS-LFCS-91-180, Lab-

oratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1991

[Weg88] Peter Wegner. *The Object-Oriented Classification Paradigm.* In *Research Directions in Object-Oriented Programming* ed Bruce Shriver and Peter Wegner. The MIT Press 2nd edition 1988



University of
Hertfordshire

CCS and Object-Oriented Concepts

M. Buchanan and R. G. Dickerson

July 1992

School of Information Sciences
Division of Computer Science
Technical Report No.140

CCS and Object-Oriented Concepts

Mary Buchanan and Bob Dickerson
Division of Computer Science, University of Hertfordshire
College Lane, Hatfield, Herts. AL10 9AB

July 1992

Abstract

The viability of using CCS as a formal specification language for classes of objects is investigated. The class based object-oriented paradigm is assumed throughout. It is concluded that CCS can be used to specify classes of objects and that it is particularly well suited for describing classes in which the time-ordering of operations is important. Further work is needed to evaluate the use of CCS to describe a complete system.

Sub-type inheritance can be expressed in CCS but at the expense of added complexity. Restriction inheritance can be expressed simply and clearly.

1 Introduction

CCS [Mil89] is a theory of communicating systems. "People will use it only if it enlightens their design and analysis of systems; therefore the experiment is to determine the extent to which it is *useful*, the extent to which the design process and analytical methods are indeed improved by the theory." [Mil89 page1]. This paper describes an experiment in which CCS is used to specify classes of objects and inheritance in the object-oriented paradigm.

2 Objects and Classes

2.1 Objects

An object has a set of operations and a state which remembers the effect of the operations; it also has a unique identity. Since the state of an object can be manipulated only by the operations exported via the object's interface, the details of the internal implementation of the state are hidden from the external view of the object. To define an object's interface it is necessary to consider the input events (the stimuli to which the object must react) and the output events (the responses the object should give).

2.2 Classes

In class-based object-oriented terminology, objects are considered to be instantiations of a class. A class can be considered to be a template describing the state and behaviour which all objects of the class have in common. A class also encompasses the concept of type. The set of operations declared in the class defines the interface of objects of the class; hence a class defines an abstract data type which can be used to determine the type compatibility of objects. Since many uniquely identifiable objects can be derived from a class, a class can also be considered to define a set of objects.

Although a class can be viewed as an implementation of an abstract data type, a class can be more than this. Whereas an abstract data type describes services provided by the type, a class can also describe services required from other classes. A class also tends to be a module and therefore to be separately compilable.

2.3 CCS and classes of objects

The existence of state within an object means that the order in which the object's operations are invoked can be important. Each object can therefore be regarded as an independent machine which can be described in terms of an equivalent state machine [Boo91 page 83]. Entity-life histories are useful to capture the time-ordering of actions in an informal manner [Buc90, Bri91, Dav90]. However, entity life histories have to be viewed in isolation since there is no way of composing components or of deriving the effects of such composition on the resultant system. We aimed to ascertain how useful CCS would be both to capture formally the time-ordering of operations on objects and to determine the effects of composing objects to form systems. In particular we wanted to establish whether CCS could be used to model inheritance.

In CCS the specification of state and the operations on that state are encapsulated together in agents which have their own identity. The behaviour of agents consists of discrete actions. A process is an independent agent which interacts with its environment solely by communication through its input and output ports. This seems analogous to the object-based concept [Weg88] in which an object has a set of operations and a remembered state which is accessed and amended by the operations alone. The *internal* behaviour of an object is hidden such that if two objects have the same external behaviour but different internal behaviour then, in an object-oriented system, either object can be substituted for the other without the behaviour of the system changing. (Here we are considering behaviour in terms of perceived functionality, not in terms of the time taken to achieve that functionality.)

A CCS agent can be expressed at different levels of abstraction; either in terms of its interaction with the environment or in terms of its composition from other agents. This is useful for system development since we can express the desired behaviour of a system as a single agent and also as the combination of simpler agents.

In CCS, the concept of *observation equivalence* expresses the equivalence of two processes which as stand-alone processes can perform the same pattern of external communications, but whose internal behaviour may differ. However, the observation equivalence of two processes, P and Q, is not enough to ensure that Q may be substituted for P in a system in which P is a component. The reason for this is that observation equivalence is not preserved by summation [Mil89 page 112]; summation is used in CCS to specify a choice of actions. Consider the expression

$$\tau.a \approx a$$

CCS describes the time-ordering of events but there is no sense of elapsed time. As a result, the fact that we might have to wait for the silent action τ to occur before we can perform the action a is of no significance. The expression τ.a means that we can perform the action a and so it is observation equivalent to the action a alone. However, consider the expression

$$a + b$$

Here, we can always perform the action a or the action b. From the above, $\tau.a \approx a$ but we cannot substitute $\tau.a$ for a in $a + b$.

$$\tau.a + b \not\approx a + b$$

If a τ action occurs, then the next action to occur must be an a. In other words we no longer have the choice between actions a or b because the τ action has precluded the b action.

For substitution to be possible we need congruence. Congruence is achieved if we have both observation equivalence and *stability*. A process P is stable if neither P itself nor any derivative of P has a leading τ action. Without stability we have the possibility that a τ action may cause a desired choice of actions to be overidden internally by the system such that choice is denied and one action is enforced. Pre-emptive 'τ's have proved troublesome in the work we describe on inheritance.

3 The Specification of Classes

We consider the viability of using CCS to specify classes of objects which have been specified informally using entity life histories. We have adopted the convention, used in Jackson System Development [Jac83], that the state of an entity can be read at any time and that this is assumed rather than specified explicitly.

3.1 Identity Cards

We shall consider a system in which a class Card is required. An object of class Card is used for identification purposes and when created it will have a card number. A card can be allocated, that is an identification for the owner of the card will be added to the card, and deallocated when the identification will be removed. At the end of its life, a card is deleted from the system. Figure 1. shows an entity life history depicting the class Card [Bri91].

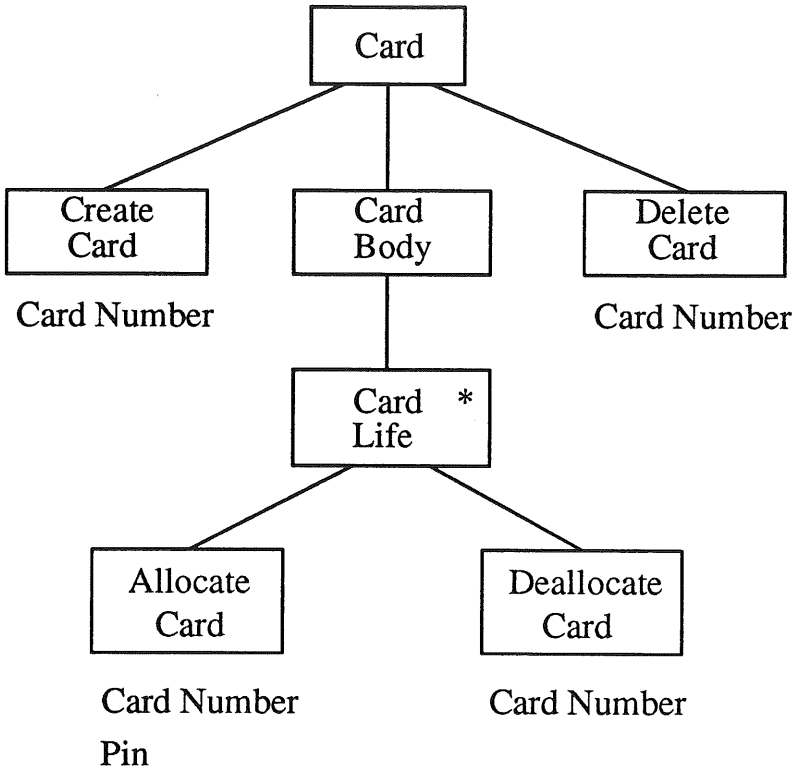


Figure 1: The Card Entity Life History

The Card class can be described in CCS as:

$$Card \stackrel{def}{=} createCard.Card1$$
$$Card1 \stackrel{def}{=} deleteCard.0 + allocateCard.deallocateCard.Card1$$

The CCS specification of the class Card is obviously concise and more importantly, it enforces the time-ordering of actions which is necessary for the particular class Card. Once a particular card has been created, it is in a new state ($Card1$) and in this state it cannot be created again. In state $Card1$, a card can perform the action *deleteCard* whereupon it reaches the state **0** from which it is incapable of further action, that is the card is no longer in the system. Alternatively, a card in state $Card1$ can perform the action *allocateCard* after which the only possible action is *deallocateCard* at which point the card is back in state $Card1$.

The CCS specification of card describes the operations or actions explicitly. However, the attributes or state variables of the card are implicit. The *createCard* action will result in a card having a card number. When *allocateCard* happens, the card attributes will be card number and some form of personal identity number (a PIN) and/or name, department or whatever is deemed appropriate for the system in hand. Since the attributes are implied, there is nothing to show what the attributes are. It is envisaged that such an abstract view of the data might cause problems in a specification aimed at object-oriented design.

We can add state variables (attributes) to the CCS specification by using the value passing features of the calculus:

$$Card \stackrel{def}{=} createCard(cn).Card1(cn)$$
$$Card1(cn) \stackrel{def}{=} deleteCard(cn).0 + allocateCard(cn, pn).deallocateCard(cn, pn).Card1(cn)$$

where the state variables are:

- *cn* a card number
- *pn* a personal identification number.

Card is an agent in a state in which communication can occur at the port *createCard* such that a value (from the set CardNumber) is received and becomes the value of the variable *cn*. Similarly, in state $Card1(cn)$ a value (from the set PIN) can be received at the port *allocateCard* and becomes the value of the variable *pn*.

The variables in the equations are given scope in two ways. In the first, the variable in an input prefix, such as '$createCard(cn).$' , has scope in the agent expression which begins with the prefix. In the second way, the variable on the left-hand side of a defining equation has scope over the whole equation. For example, the variable *cn* has scope over the whole equation

$$Card1(cn) \stackrel{def}{=} deleteCard(cn).0 + allocateCard(cn, pn).deallocateCard(cn, pn).Card1(cn)$$

whereas the the variable *pn* is in an input prefix and has scope over the agent expression

$$allocateCard(cn, pn).deallocateCard(cn, pn).Card1(cn)$$

It should be emphasised that the value passing calculus is only an abbreviation for having a complete set of equations for each discrete card in the system:

$$\begin{aligned} Card &\stackrel{def}{=} \sum_{c \in C} \sum_{p \in P} createCard_c.Card1_c \\ Card1_c &\stackrel{def}{=} deleteCard_c.0 + allocateCard_{c,p}.deallocateCard_{c,p}.Card1_c \end{aligned}$$

where

- C is the set of all values of type CardNumber and c is a member of C
- P is the set of all values of type Pin and p is a member of P.

Card can thus be interpreted as the collection (or class) of all the individual card objects in the system.

We have tended not to use the value passing calculus since the concurrency work bench [Cle89] cannot be used to test transitions having parameters and also because the existence of the parameters increases the complexity of the equations.

3.2 Summary

To summarise, the CCS specification can be considered to describe the class *Card*. The operation *createCard* would be requested from the class to create a card. The other actions describe the behaviour which all objects of the class share. The required time-ordering of actions is enforced.

4 Inheritance

Inheritance is one of the distinguishing features of the object-oriented paradigm. The different interpretations given to the meaning of inheritance depend largely on whether inheritance is being used to achieve sub-type hierarchies or to effect code-sharing without a type relationship.

4.1 Inheritance and the sub-type relationship

A type, the sub-type or descendant, can be derived from another type, the super-type or ancestor, such that the sub-type is a more specialised form of the super-type. In strict inheritance the sub-type will inherit all the attributes and operations of the super-type and will have additional attributes and/or operations. For example, the super-type *Card* could be inherited by a sub-type *CardPlus* containing the additional operation *changePin* to enable the pin number to be changed. Since *CardPlus* has all the properties of the parent class *Card*, it is a sub-type of *Card* and an instance of *CardPlus* can be used wherever an instance of *Card* is expected. The relationship between the instances of the types is referred to as an *is-a* relationship, we can say that an instance of type *CardPlus* is-a instance of type *Card*. The is-a relationship is transitive, hence if another type *CardPlus+* was created by inheritance from *CardPlus*, values of the type *CardPlus+* would stand in an is-a relationship to *Card* as well as to *CardPlus*. The is-a relationship is not symmetric and we cannot say that a *Card* is-a *CardPlus*.

We would want users (clients) of the sub-type inheritance hierarchy to be aware of the inheritance structure and the relationships between the ancestors and descendants. A descendant is always a more specialised form of a more general ancestor.

The interface of a class defines the specification for the class such that the profiles of the exported operations are declared. A class conforms to another if it can be used in all contexts where the other class is expected. Conformance depends only on the interfaces of the classes and not on the implementation; the interface of a class subsumes the interface of any class to which it conforms. For sub-type inheritance, it is essential that the sub-class inherits the interface of the super-class whereas the implementation may or may not be inherited. In languages such as Trellis/Owl and Emerald [Atk91 page 15] conformance is used as the sole basis for defining type compatibility and such languages could therefore support sub-type inheritance of interfaces only. However, in Eiffel classes must have implementations as well as interfaces related by inheritance if the classes are to be type-compatible.

4.2 Inheritance and the like relationship

4.2.1 Restriction

In this interpretation of inheritance, simpler less specialised classes can be created from more complex specialised classes. If, for example, the *CardPlus* class existed and there was a requirement to create a *Card* class, then this could be achieved by inheriting the code from the *CardPlus* class and restricting the *changePin* operation. We have reversed the inheritance structure from that described in the previous section on sub-typing since now we have that *CardPlus* is the super-class and *Card* is the sub-class. The relationship between the super-class and sub-class can be viewed as a *like* relationship. In the example given, we could say that a *Card* is like a *CardPlus* but we would not be able to use an instance of the sub-class *Card* wherever an instance of the class *CardPlus* was expected. However, the super-class *CardPlus* is a sub-type of the sub-class *Card* and languages such as Emerald and Trellis/Owl would presumably allow the assignment of an instance of class *CardPlus* to an instance of class *Card* since *CardPlus* conforms to *Card*.

If the like relationship is to hold when using inheritance and restriction, we again have that the interface must be inherited but that the implementation need not be.

4.2.2 Restriction and Enrichment

If a new class is derived via inheritance such that not only are operations in the super-class restricted but also new operations are added to the sub-class, then the sub-class may still have a *like* relationship to the super-class but the sub-class will no longer be a simpler, less specialised

version of the super-class. Consequently, the super-class will no longer be a sub-type of the sub-class. For example, if a class *AnotherCard* inherits from class *CardPlus* such that the operation *changePin* is restricted and another operation, such as *addAddress*, is added to the new class, then *AnotherCard* is still like *CardPlus* but *CardPlus* is not a sub-type of *AnotherCard*.

To maintain a like relationship, we again require that the interface is inherited whether or not the implementation is inherited.

4.3 Inheritance without a semantic relationship

It is possible to use inheritance for the sole purpose of reusing the code of an existing class in a new class; there need not be a semantic relationship between the classes. If, for example, a class Dequeue existed in which a double ended queue was defined, then this could be inherited by a class, Stack, to define a stack. All of the Dequeue code would be inherited by the class Stack, but the operation to enable an item to join the back of the queue would be restricted in order to prevent an item from being added to the bottom of a stack. The dequeue operations would probably be renamed to those more commonly used for stacks, for example *head* would be renamed *top*. A stack is not like a dequeue and there is no type compatibility between the classes; the inheritance relationship should therfore be hidden from clients.

For this type of inheritance, it is essential that the implementation code is inherited but it is unlikely that the interface will be inherited.

4.4 Inheritance and specification

As regards specification we can view the potential benefits of inheritance from two perspectives. Conceptually, it will be easier to understand a specification in which sub-type inheritance is used as an abstraction to describe specialisation-generalisation relationships which exist between classes. Once the behaviour of a base class has been understood, it is necessary only to understand the new behaviour added. "Inheritance may be applied to explicitly express commonality, beginning with the early activities of analysis" [Coa91]. Pragmatically, if extra capability is required for the same or for a new class after a specification has been written, it is attractive to have to describe only the new behaviour as an extension of the old and not to have to completely respecify.

Similarly, if there is a need to create a simpler version of a class (such as a *Card* from a *CardPlus*) where there is a semantic relationship between the classes, then it could be an advantage to use inheritance because it is only necessary to understand what behaviour has been removed.

However, it would seem undesirable to use inheritance in specifications merely to reuse code without any semantic relationship between the classes. Great care would be needed in order to convey the intended change of semantics between the classes and it would seem likely that greater clarity could be obtained by other means.

5 CCS and sub-type inheritance

In this paper we consider inheritance mainly from the sub-type point of view such that *CardPlus* is a sub-type of the more general class *Card*. We investigate the extent to which CCS can model such inheritance.

5.1 A new class defined to extend the behaviour of class Card

Suppose we want to define another class, *CardPlus* having the same behaviour as *Card* but with the additional operation that the PIN can be changed. An entity life history for *CardPlus* is shown in Figure 2.

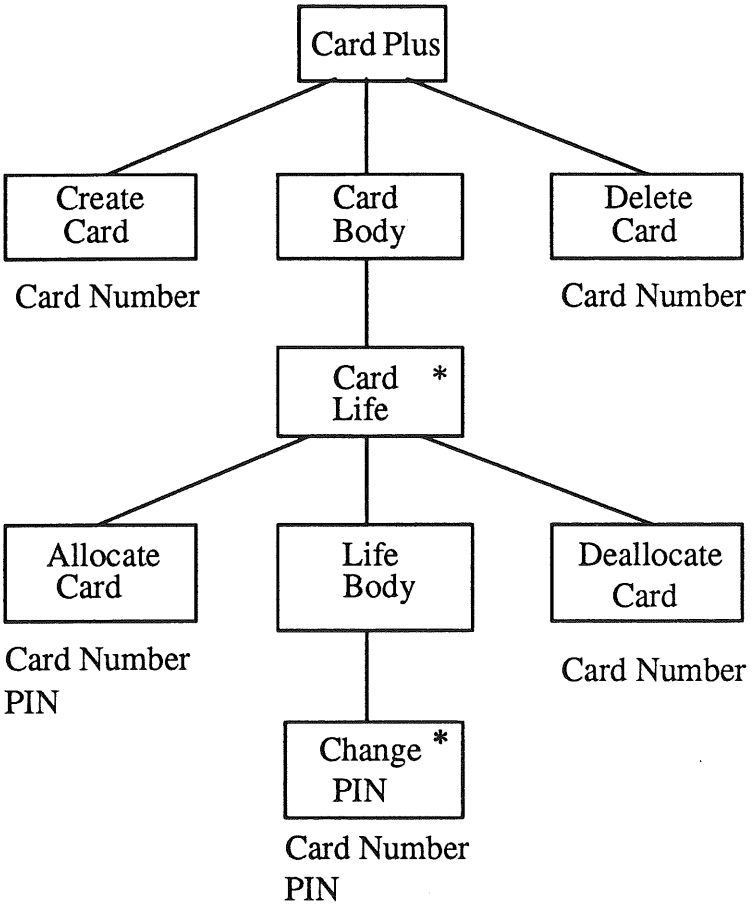


Figure 2: The CardPlus Entity Life History

The CCS specification for *CardPlus* is:

$$
\begin{aligned}
CardPlus &\stackrel{def}{=} createCard.CardPlus1 \\
CardPlus1 &\stackrel{def}{=} deleteCard.0 + allocateCard.CardPlus2 \\
CardPlus2 &\stackrel{def}{=} changePin.CardPlus2 + deallocateCard.CardPlus1
\end{aligned}
$$

From this specification, we can see that the *changePin* action can only be applied after a card has been allocated and before it is deallocated but that within these constraints, the PIN can be changed as often as wanted or it need never be changed.

In order to simplify the following discussion, we omit the *createCard* and *deleteCard* actions. The adapted *Card* class, *CardA*, is defined as:

$$CardA \stackrel{def}{=} allocateCard.deallocateCard.CardA$$

and similarly, the adapted CardPlus class is defined as:

$$CardPlusA \stackrel{def}{=} allocateCard.CardPlusA1$$
$$CardPlusA1 \stackrel{def}{=} changePin.CardPlusA1 + deallocateCard.CardPlusA$$

5.2 Inheriting the behaviour of class Card

We aim to establish the viability of using CCS to represent inheritance. We want to ascertain whether it is possible to form a new class, $CardH$, which inherits behaviour from $CardA$ and has the added behaviour that the PIN can be changed. $CardH$ is to be a sub-type of the super-type $ClassA$ and is to be congruent with $CardPlusA$ so that in a given specification the two would be interchangeable.

The only way that agents can communicate with one another is via their ports. Accordingly, we define a new agent which is congruent with $CardA$ but which has the potential for communication through ports other than $allocateCard$ and $deallocateCard$.

Consider the agents:

$$CardC \stackrel{def}{=} allocateCard.\overline{a}.b.CardC$$
$$CardD \stackrel{def}{=} a.deallocateCard.\overline{b}.CardD$$

If we compose $CardC$ and $CardD$, and restrict the ports a and b so that only internal communication is possible via these ports, we have

$$CardCD \stackrel{def}{=} (CardC|CardD)\backslash\{a,b\}$$

Figure 3 shows $CardCD$ in diagrammatic form.

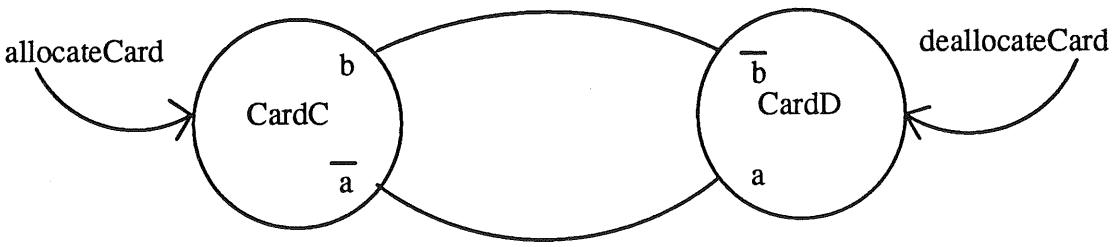


Figure 3: The CardCD

We can use the expansion theorem [Mil89 page 69] to examine the behaviour of the concurrent (i.e. interleaved) processes,

$$\begin{aligned}
CardCD &= (CardC|CardD)\backslash\{a,b\} \\
&= (allocateCard.\overline{a}.b.CardC|a.deallocateCard.\overline{b}.CardD)\backslash\{a,b\} \\
&= (allocateCard.(\overline{a}.b.CardC|a.deallocateCard.\overline{b}.CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.(b.CardC|deallocateCard.\overline{b}.CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.deallocateCard.(b.CardC|\overline{b}.CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.deallocateCard.\tau.(CardC|CardD))\backslash\{a,b\} \\
&= (allocateCard.\tau.deallocateCard.\tau.CardCD)\backslash\{a,b\}
\end{aligned}$$

By the τ law $a.\tau.P = a.P$ (which holds for observation equivalence), the silent action τ can be be ignored when it occurs between actions, hence

$$CardCD \quad = allocateCard.deallocateCard.CardCD$$

and since

$$CardA \quad = allocateCard.deallocateCard.CardA$$

we can observe that $CardA$ and $CardCD$ are equal and so must be congruent. The significance of this is that if in a system we have $CardA$, we can substitute $CardCD$ for $CardA$. We shall use $CardCD$ to try and establish if we can use it in an inheritance relationship. We want to define a new agent having the *changePin* behaviour in such a way that we can compose it with $CardCD$ such that we have congruence with $CardPlusA$.

Consider $CardE$ defined as:

$$\begin{aligned} CardE &\stackrel{def}{=} a.CardE1 \\ CardE1 &\stackrel{def}{=} changePin.CardE1 + \overline{c}.CardE \end{aligned}$$

Now in order to link $CardE$ in to $CardCD$, we relabel the a port in $CardD$:

$$CardDa \stackrel{def}{=} CardD[c/a]$$

Finally, we define $CardH$ as:

$$CardH \stackrel{def}{=} (CardC|CardDa|CardE)\backslash\{a, b, c\}$$

Figure 4 shows $CardH$ in diagrammatic form.

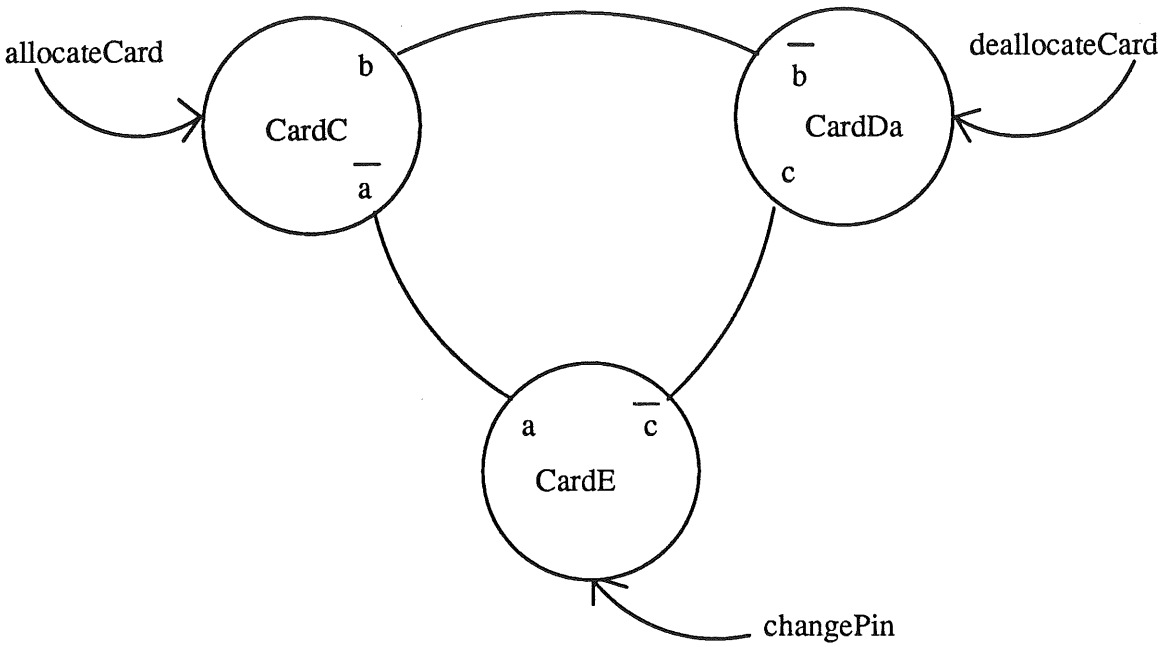


Figure 4: The CardH

However, *CardH* does not exhibit the desired behaviour. Testing on the concurrency workbench [Cle89] shows that *CardH* and *CardPlusA* are not observation equivalent. The concurrency workbench can also test for preorder relationships as defined in [Hen88]. Thus we have that

$$CardH \approx_{may} CardPlusA$$

which means that *CardH* and *CardPlusA* *may* (not must) be equivalent.

Although the traces of the alternative sequences of actions which can be undertaken by *CardH* and *CardPlusA* are the same, this only tells us that it is possible for *CardH* and *CardPlusA* to follow the same sequences of actions, not that they must be able to follow such sequences at all times.

We also have that

$$CardH \sqsubseteq_{must} CardPlusA$$

This tells us that there are no actions which *CardH* can perform that *CardPlusA* cannot also perform but tells us nothing about the extra actions that *CardPlusA* can perform over and above tthe actions of *CardH*. For *CardH* to be able to simulate *CardPlusA* , we would need to have observation equivalence or failing that we would want to have that the behaviour of *CardH* was a super-set of the behaviour of *CardPlusA*, that is $CardPlusA \sqsubseteq_{must} CardH$; however testing shows that this is false.

When we consider the expansion theorem, we discover that *CardPlusA* must allow one to *allocateCard*, *changePin* and *deallocateCard* whereas *CardH* must allow one to *allocateCard* and *deallocateCard* but that it may or may not allow one to *changePin*.

Thus by the expansion theorem,

$$
\begin{aligned}
CardH &\stackrel{def}{=} (CardC|CardDa|CardE)\backslash\{a,b,c\} \\
CardH &= (allocateCard.\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.CardE1)\backslash\{a,b,c\} \\
CardH &= (allocateCard.(\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.cardE1))\backslash\{a,b,c\} \\
CardH &= (allocateCard.\tau.(b.CardC|c.deallocateCard.\overline{b}.CardD| \\
&\qquad (changePin.CardE1 + \overline{c}.CardE)))\backslash\{a,b,c\} \\
CardH &= (allocateCard.\tau.(\tau.(b.CardC|deallocateCard.\overline{b}.CardD|CardE) \\
&\qquad + \\
&\qquad changePin(b.CardC|c.deallocateCard.\overline{b}.CardD|CardE1))) \\
&\qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

Now we can see where the problem lies. The silent action after the *allocateCard* action is of no concern but it is followed by a choice of actions, the first of which has a leading τ. If the leading τ occurs , then we no longer have a choice of action between the *changePin* and the *deallocateCard* actions since the *changePin* action will have been pre-empted. Taking the expansion one stage further makes this clearer.

$$
\begin{aligned}
CardH &= (allocateCard.\tau.(\tau.deallocateCard(b.CardC|.\overline{b}.CardD|CardE) \\
&\qquad + \\
&\qquad changePin.(\tau.(b.CardC|deallocateCard.\overline{b}.CardD|a.CardE1) \\
&\qquad\qquad + \\
&\qquad\qquad changePin(b.CardC|c.deallocateCard.\overline{b}.CardD| \\
&\qquad\qquad\qquad a.CardE1))))\backslash\{a,b,c\}
\end{aligned}
$$

Once the leading τ has occurred, via an autonomous action of the system, then the next action which can be taken from outside the system must be a *deallocateCard* regardless of whether this is the action actually wanted.

5.3 The problem with CardH

The problem of the silent τ action is caused by the agent $CardE$ defined as:

$$
\begin{aligned}
CardE &\stackrel{def}{=} a.CardE1 \\
CardE1 &\stackrel{def}{=} changePin.CardE1 + \overline{c}.CardE
\end{aligned}
$$

The presence of $\overline{c}.CardE$ in $CardE1$ lies at the heart of the matter. The $\overline{c}$ means that silent and secret communication can occur with $CardDa$. Such an action is beyond the control of the environment in which $CardH$ finds itself so the behaviour of $CardH$ is unpredictable. If the silent communication with $CardDa$ occurs then the *changePin* action is denied to the environment, without the environment being aware that this is so.

5.4 A Partial Solution

Various attempts were made to overcome the problem of the unwanted pre-emptive τ but without success. However, we were able to arrive at a partial solution; the solution is partial because it leads to re-specification of the original system. In order to prevent the unwanted silent communication between $CardE$ and $CardDa$, we introduce an external action such that the environment has to actively request that it wants to deallocate a card *before* actually activating the *deallocateCard* action. In order to do this, $CardE$ is replaced by $CardF$ defined as:

$$CardF \stackrel{def}{=} a.CardF1$$
$$CardF1 \stackrel{def}{=} changePin.CardF1 + requestDeallocate.\overline{c}.CardF$$

The effect of the *requestDeallocate* action is to prevent the unwanted silent communication via the $\overline{c}$. After an external *requestDeallocate* action has been chosen, a wanted silent communication is enabled and has the desired effect that the next permitted action is *deallocateCard*. Consider *CardI* as shown diagrammatically in Figure 5.

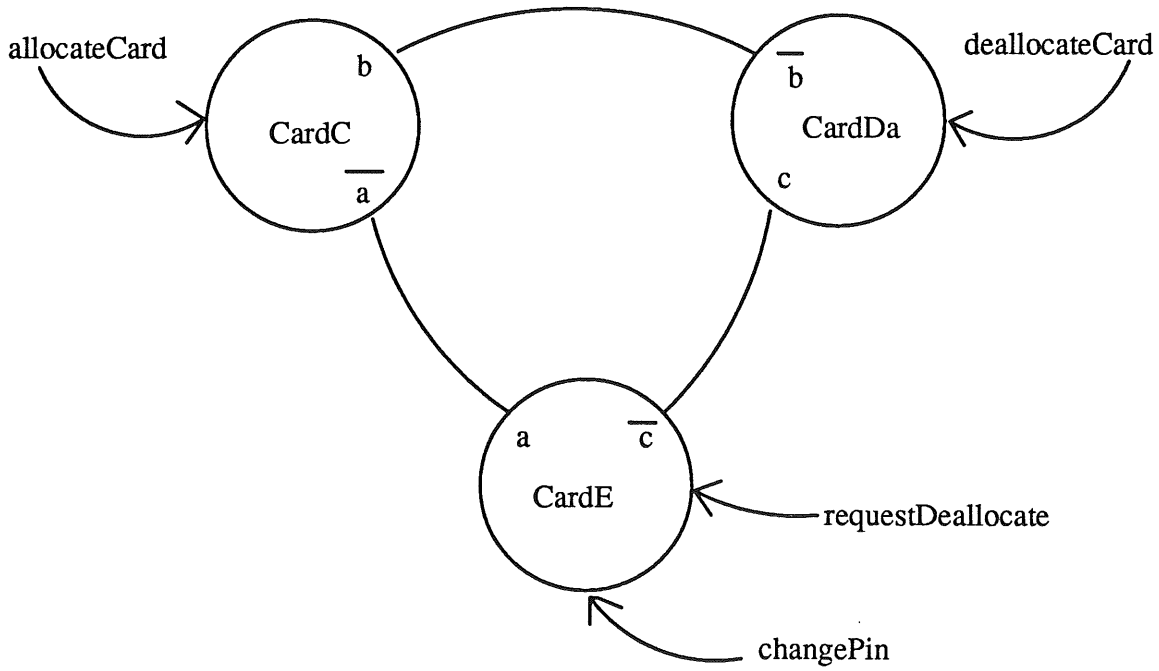


Figure 5: The CardI

CardI is defined in CCS as:

$$CardI \stackrel{def}{=} (CardC|CardDa|CardF)\backslash\{a,b,c\}$$

By the expansion theorem,

$$\begin{aligned}
CardI &= (allocateCard.\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.cardF1)\backslash\{a,b,c\}\\
CardI &= (allocateCard.(\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.cardF1))\backslash\{a,b,c\}\\
CardI &= (allocateCard.\tau.(b.CardC|c.deallocateCard.\overline{b}.CardD|\\
&\qquad (changePin.CardF1 + requestDeallocate.\overline{c}.CardF)))\backslash\{a,b,c\}\\
CardI &= (allocateCard.\tau.(changePin.(b.CardC|c.deallocateCard.\overline{b}.CardD|CardF1)\\
&\qquad +\\
&\qquad requestDeallocate.(b.CardC|c.deallocateCard.\overline{b}.CardD|\overline{c}.CardF)))\\
&\qquad\qquad \backslash\{a,b,c\}
\end{aligned}$$

We can now see that it is only after a *requestDeallocate* that *CardDa* and *CardF* can communicate via a silent action as shown below.

$$
\begin{aligned}
CardI \;=\; & (allocateCard.\tau. \\
& \quad (changePin. \\
& \qquad (changePin.(b.CardC|c.deallocateCard.\overline{b}.CardD|CardF1) \\
& \qquad + \\
& \qquad requestDeallocate.(b.CardC|c.deallocateCard.\overline{b}.Card| \\
& \qquad\qquad \overline{c}.CardF)) \\
& \quad + \\
& \quad requestDeallocate.\tau.(b.CardC|deallocateCard.\overline{b}.CardD|CardF))) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

$$
\begin{aligned}
CardI \;=\; & (allocateCard.\tau. \\
& \quad (changePin. \\
& \qquad (changePin. \\
& \qquad\quad (changePin(b.CardC|c.deallocateCard.\overline{b}.CardD| \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad CardF1) \\
& \qquad\quad + \\
& \qquad\quad requestDeallocate.(b.CardC|c.deallocateCard.\overline{b}. \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad CardD|\overline{c}.CardF)) \\
& \qquad + \\
& \qquad requestDeallocate.\tau.(b.CardC|deallocateCard.\overline{b}.CardD| \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad CardF)) \\
& \quad + \\
& \quad requestDeallocate.\tau.deallocateCard.(b.CardC|\overline{b}.CardD|a.CardF1))) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

$$
\begin{aligned}
CardI \;=\; & (allocateCard.\tau. \\
& \quad (changePin. \\
& \qquad (changePin. \\
& \qquad\quad (changePin. \\
& \qquad\qquad (changePin(b.CardC|c.deallocateCard.\overline{b}. \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad CardD|CardF1) \\
& \qquad\qquad + \\
& \qquad\qquad requestDeallocate.(b.CardC| \\
& \qquad\qquad\qquad\qquad c.deallocateCard.\overline{b}.CardD|\overline{c}.CardF)) \\
& \qquad\quad + \\
& \qquad\quad requestDeallocate.\tau.(b.CardC|deallocateCard.\overline{b}. \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad CardD|CardF)) \\
& \qquad + \\
& \qquad requestDeallocate.\tau.deallocateCard.(b.CardC|\overline{b}.CardD| \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad a.CardF1)) \\
& \quad + \\
& \quad requestDeallocate.\tau.deallocateCard.\tau.(CardC|CardD|CardF))) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \backslash\{a,b,c\}
\end{aligned}
$$

By defining $CardI$, we have succeeded in removing the instability in $CardH$ caused by the leading τ. However, $CardI$ does not match the specification of $CardPlusA$ due to the presence of the extra action, *requestDeallocate*. $CardI$ is in fact congruent with $CardPlusF$ defined as:

$$
\begin{aligned}
CardPlusF &\stackrel{def}{=} allocateCard.CardplusF1 \\
CardPlusF1 &\stackrel{def}{=} changePin.CardPlusF1 + requestDeallocate.deallocateCard.CardPlusF
\end{aligned}
$$

We can regard $CardPlusF$ as a more friendly version of $CardPlusA$ in that the *requestDeallocate*

could be interpreted as giving a user a warning message such as "Are you sure you want to deallocate this card? Such an action will result in the card having to be reallocated".

Tests on the concurrency workbench confirm that $CardI$ is congruent with $CardPlusF$. $CardI$ can be considered to have inherited the behaviour of $CardCD$ and to have extended the behaviour to include the *changePin* and *requestDeallocate* operations. If in a system we had $CardA$ and wanted to inherit from it to form a class having the behaviour of $CardPlusF$, then we could replace $CardA$ with $CardCD$ (since they are congruent) and use $CardCD$ as the supertype from which to derive the subtype $CardI$. Since $CardI$ is congruent with $CardPlusF$, $CardI$ has the behaviour required of the new class.

5.5 A Single ChangePin Action

It should be stressed that the problems caused in $CardH$ by the unwanted pre-emptive τ arose because we wanted to be able to change the PIN more than once. No such problems arise if one specifies that the system must change the PIN once and once only. Consider such a card defined as:

$$SimpleCardPlus \stackrel{def}{=} allocateCard.changePin.deallocateCard.SimpleCardPlus$$

We can define $CardG$ as:

$$CardG \stackrel{def}{=} a.changePin.\overline{c}.CardG$$

Then $CardCDG$ is:

$$CardCDG \stackrel{def}{=} (CardC|CardDa|CardG)\backslash\{a,b,c\}$$

By the expansion theorem,

$$\begin{aligned}
CardCDG &= (allocateCard.\overline{a}.b.CardC|c.deallocateCard.\overline{b}.CardD|a.changePin.\overline{c}CardG) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.(b.CardC|c.deallocateCard.\overline{b}.CardD|changePin.\overline{c}.CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.(b.CardC|c.deallocateCard.\overline{b}.CardD|\overline{c}.CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.\tau.(b.CardC|deallocateCard.\overline{b}.CardD|CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.\tau.deallocateCard.(b.CardC|\overline{b}.CardD|CardG)) \\
&\qquad \backslash\{a,b,c\} \\
CardCDG &= (allocateCard.\tau.changePin.\tau.deallocateCard.\tau.(CardC|CardD|CardG)) \\
&\qquad \backslash\{a,b,c\}
\end{aligned}$$

Since all the τ are non-leading, they can be disregarded and $CardCDG$ is seen to be congruent with $SimpleCardPlus$.

6 Interrupts

Unfortunately, CCS has no way to model interrupts. Using the notation P^iQ to represent P interrupted by Q, we have that P^iQ behaves like P until Q does anything at all whereupon it behaves like Q. An interrupt operator might have enabled us to solve the problem in $CardH$

by allowing the *deallocateCard* operation to interrupt $CardH$ such that $CardH$ returned to its original state. Consider $CardH_2$ defined as:

$$CardH_2 \stackrel{def}{=} ((CardC|CardE_2)\backslash\{a,b\})^{i} deallocateCard.CardH_2$$

where

$$\begin{aligned} CardE_2 &\stackrel{def}{=} a.CardE_22 \\ CardE_22 &\stackrel{def}{=} changePin.CardE_22 \end{aligned}$$

$CardE_2$ is the same as $CardE$ except that the c link with $CardD$ has been removed. The behaviour of $CardH_2$ will be that of $(CardC|CardE_2)\backslash\{a,b\}$ until a *deallocateCard* action occurs and the original state is resumed. However, the specification is still not quite right since a *deallocateCard* can occur before an *allocateCard*. To overcome this, we define $CardH_3$ as:

$$CardH_3 \stackrel{def}{=} allocateCard.\left(((\overline{a}.b.CardC|CardE_2)\backslash\{a,b\})^{i} deallocateCard.CardH_3\right)$$

Although such a specification would give us the behaviour we want, we are perhaps moving away from inheritance in that we are only using $CardC$ and not $CardD$. Instead of inheriting all the behaviour of $CardCD$, we are only inheriting the behaviour of $CardC$.

7 The π-Calculus

The π-Calculus [Mil91] enables communication between agents to carry information which changes the linkage between the agents and thereby can describe agents which have a changing structure. This would seem to offer a means of expressing inheritance.

The major advance over CCS is the ability to send the names of links as parameters in communications. A link is formed between agents having complementary labels to ports. No distinction is made between link names, variables and ordinary data values; they are all just *names*. Thus there are only two essential classes of entity: names and agents.

"It is considered that the π-calculus will lead to a better understanding of object-oriented programming" [Mil91 Page2].

7.1 A First Attempt

We want to define our basic $Card$ so that it can receive new actions. Consider $CardB$ defined as:

$$\begin{aligned} CardB &\stackrel{def}{=} allocateCard.CardB1 \\ CardB1 &\stackrel{def}{=} a(x).x.CardB1 + deallocateCard.CardB \end{aligned}$$

$a(x).P$ means that agent P can receive a name z (of a link or of a value) at port a and then behaves as $P\{z/x\}$, that is all parameters x in P are replaced by the actual name z. Thus a new link x can be sent to $CardB$ along the link a, then the new link can be used. In the case where we want a *changePin* action sent to $CardB$, we can define:

$$CardS \stackrel{def}{=} \overline{a}changePin.CardS$$

$\overline{a}t$ is interpreted to mean transmit the value t along the link a.

We then compose as:

$$CardBS \stackrel{def}{=} (a)(CardB|CardS)$$

Restriction in the π-calculus is expressed as $(a)P$ meaning that external actions at the ports a and $\overline{a}$ are prohibited but internal communications along a are permitted for the components of P.

Unfortunately, when we apply the expansion theorem to $CardBS$, we discover that a leading τ occurs such that the *deallocateCard* action can be pre-empted.

In order to overcome the pre-emptive τ, we define $CardR$ such that there is a guard (*newAction*) on the $a(x)$ as:

$$\begin{aligned} CardR &\stackrel{def}{=} allocateCard.CardR1 \\ CardR1 &\stackrel{def}{=} newAction.a(x).x.CardR1 + deallocateCard.CardR \end{aligned}$$

(We considered putting the guard, *sendChangePin*, on the $\overline{a}$ in $CardS$ but this caused problems such as the possibility of a *sendChangePin* after a *deallocateCard*.)

$$CardS \stackrel{def}{=} \overline{a}changePin.CardS$$

By composition,

$$CardRS \stackrel{def}{=} (a)(CardR|CardS)$$

By expansion,

$$\begin{aligned}
CardRS &= (a)(allocateCard.CardR1|\overline{a}changePin.CardS) \\
CardRS &= allocateCard.(a)(CardR1|\overline{a}changePin.CardS) \\
CardRS &= allocateCard.(a)((newAction.a(x).x.CardR1 \\
&\qquad + \\
&\qquad deallocateCard.CardR)|\overline{a}changePin.CardS) \\
CardRS &= allocateCard.(a)((newAction.a(x).x.CardR1|\overline{a}changePin.CardS) \\
&\qquad + \\
&\qquad deallocateCard.CardR|\overline{a}changePin.CardS) \\
CardRS &= allocateCard.(a)((newAction.(a(x).x.CardR1|\overline{a}changePin.CardS) \\
&\qquad + \\
&\qquad deallocateCard.(allocateCard.CardR1|\overline{a}changePin.CardS)) \\
CardRS &= allocateCard.(a)(newAction.\tau.(changePin.CardR1|CardS) \\
&\qquad + \\
&\qquad deallocateCard.allocateCard.(CardR1|\overline{a}changePin.CardS)) \\
CardRS &= allocateCard.(a)(newAction.\tau.changePin.(CardR1|CardS) \\
&\qquad + \\
&\qquad deallocateCard.allocateCard.(CardR1|\overline{a}changePin.CardS))
\end{aligned}$$

We can see that we have the behaviour we require in that after an *allocateCard* action has occurred, we can now perform the *changePin* action as an alternative choice to a *deallocateCard* action. Furthermore, a *changePin* action can never be performed immediately after a card has been deallocated.

However, whenever we wish to perform a *changePin* action we have to go through the *newAction* action and send the *changePin* action to $CardR$. This seems unnecessarily laborious. What we really want is to send the *changePin* action to the $CardR$ once only and then use it repeatedly as required.

7.2 First refinement to CardRS

$$
\begin{aligned}
CardR_2 &\stackrel{def}{=} allocateCard.CardR_21 \\
CardR_21 &\stackrel{def}{=} newAction.a(x).CardR_22 + deallocateCard.CardR_2 \\
CardR_22 &\stackrel{def}{=} x.CardR_22 + deallocateCard.CardR_2
\end{aligned}
$$

$CardR_21$ enables the *deallocateCard* action to be chosen in which case an *allocateCard* action must be the next action chosen. Alternatively, *newAction* may be chosen in which case a new link x can be received. $CardR_22$ enables the x action to be performed repeatedly, without the need to use $newAction.a(x)$ each time, and also enables *deallocateCard* to be selected when required. We can compose $CardR_2$ with $CardS_2$ defined as:

$$CardS_2 \stackrel{def}{=} \overline{a}changePin.0$$

Now that we do not have to repeatedly send a *changePin* action to $CardR$, we have been able to simplify the $CardS$ definition to $CardS_2$.

By composition,

$$CardRS_2 \stackrel{def}{=} (a)(CardR_2|CardS_2)$$

7.3 Second refinement to CardRS

So far we have been discussing the case where we could add one new action to the *Card*. In order to have more flexibility, we need to be able to send more than one new action to the card $CardR$.

$$
\begin{aligned}
CardR_3 &\stackrel{def}{=} allocateCard.CardR_31 \\
CardR_31 &\stackrel{def}{=} newAction.a(x).(CardR_31\{x'/x\} + CardR_32) \\
&\quad + \\
&\quad deallocateCard.CardR_3 \\
CardR_32 &\stackrel{def}{=} x.CardR_32 + CardR_31\{x'/x\}
\end{aligned}
$$

Now it would appear that we can send two new actions to $CardR_3$ via $a(x)$ and $a(x')$. However, it is not so clear what happens with $CardR_32$; will the last action received (x') override the first action? We have not yet resolved this question and it may well be that the π-calculus can provide more elegant solutions. There is the added complexity that when new actions are added, the order in which the actions are used could well be important.

8 Data addition

Until now we have not needed to add new data to the *Card*, only new actions. Thus *changePin* was added to enable the *PIN* which was already in the *Card* to be changed. However, there are times when we might want to add new data as well as a new process. Consider the requirement that the *Card* is to contain information about the rooms that the cardholder is entitled to enter in a secure building. We need to add data about rooms, since this is not on the original card, and an action to add rooms to the original set of rooms. We will also want to be able to delete rooms from the set. We have not yet tackled these issues, but it does appear that the π-calculus may be able to address such matters.

In CCS, if one adds an operation such that a new type of data can arrive at a port, then one has implied that a corresponding new attribute has also been added to the agent.

9 Discussion

9.1 Liveness and Safety

We can say that the liveness of a system describes the desirable behaviour it must have, whereas the safety of a system describes the undesirable behaviour it must not have. We have been trying to avoid having a pre-emptive τ because the liveness of $CardplusA$ was not preserved by $CardH$ due to such a τ but there are circumstances in which a pre-emptive τ might be desirable. It might be that one wants a system to make decisions about what it did without reference to the environment. In [Bai91] the specification of a level-crossing is considered. It is required to model the situation where an observer sees either a train or cars approaching the level crossing and is never prevented from seeing either, but cannot choose between them. It would be desirable if the choice could be made internally by the system such that when, for example, an approaching train is sensed, actions are performed which will eventually allow the train to cross.

In the case of $CardH$, it might be desirable for the system to decide that for some well-defined reason the user must not be allowed to change his pin number. However, with $CardH$ we have modelled the situation in which the decision to prevent the change pin action is taken by the system purely *at random*. This is not a desirable behaviour.

9.2 CCS and Inheritance

9.2.1 Sub-type inheritance

We have had only limited success at using CCS to define sub-type inheritance and even that was achieved at the expense of adding much extra complexity to the parent *Card* class specification in the form of extra ports. In addition, we had to add the *requestDeallocate* action to the original specification in order to achieve stability.

9.2.2 Inheritance and Restriction

The restriction operator in CCS does enable inheritance to be modelled very simply, provided the new class is a restricted version of the existing class and no extra behaviour is added. If the class $CardPlusA$ was in existence and there was a requirement to provide the simpler class $NewCardA$, having the behaviour of $CardA$, then this can be achieved as:

$$NewCardA \stackrel{def}{=} CardPlusA \backslash \{changePin\}$$

$NewCardA$ is observation equivalent to $CardA$.

9.3 CCS and State

In order to show what happens to the state variables as the result of an action, it is necessary to use the value passing calculus. This has the overhead that the specifications become considerably more cumbersome, particularly if many state variables are required. There is the added disadvantage that the concurrency work bench does not handle the value passing calculus.

10 Conclusions

If inheritance is to be used in specifications, then it should aim to reduce complexity both in the semantic relationships between classes and in the specification code itself. This is particularly important when one considers that classes defined at the specification stage might not necessarily be those implemented for the final system. We have only been able to specify sub-type inheritance in CCS by introducing extra complexity into the specification. For example, in order to inherit from *CardA* we had to replace it with the more complex *CardCD*. However, inheritance can be expressed naturally and simply in CCS in the limited case where the new class is simply a restriction of the super-class.

The fact that we have had difficulty in expressing sub-type inheritance in CCS should not overshadow the benefits which CCS can bring to an object-oriented specification. The time-ordering of operations can be enforced where required. In addition, the labels of a class give all the operations for objects of the class and likewise the labels of a system give all the operations of the system. Restrictions on the labels show the operations which are internal to the system. The composition operator in the calculus makes it possible to determine the effects that concurrent objects will have on one another; as we have discovered, such effects are not always obvious from the initial specifications of the objects. However, we are not yet sure whether the effects are a reflection on the system being developed or whether they have arisen from our CCS model of the system.

The π-calculus seems to offer a much simpler means than CCS of building adaptability into a specification, although this adaptability is perhaps modelling extendibility rather than inheritance.

11 References

[Atk91] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution.* Addison-Wesley 1991

[Bai91] Jean Baillie. *A CCS Case Study: A Safety Critical System.* Software Engineering Journal, vol 6, number 4, July 1991. IEE and BCS.

[Boo91] Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings 1991

[Bri91] Carol Britton and Mary Buchanan. *Modelling Techniques for Object-Oriented Design.* Technical Report No.133, Hatfield Polytechnic, 1991

[Buc90] Mary Buchanan. *The Phantom of the Object.* MSc Project Report, Hatfield Polytechnic, 1990

[Cle89] Cleaveland, Parrow and Steffen. *The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems.* Technical Report ECS-LFCS-89-83, LFCS, Department of Computer Science, University of Edinburgh, August 1989.

[Coa91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice-Hall International 2nd edition 1991

[Dav90] N.W.Davis, M.Irving and J.E.Lee. *The evolution of object-oriented design from concept to method.* In *Managing Complexity in Software Engineering*, ed R.J.Mitchell, Peter Peregrinns Ltd on behalf of the Institution of Electrical Engineers, 1990

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* The MIT Press, 1988

[Jac83] Michael Jackson. *System Development.* Prentice Hall International, 1983

[Mil89] R.Milner. *Communication and Concurrency.* Prentice Hall International, 1989

[Mil91] Robin Milner. *The Polyadic π-Calculus: A Tutorial.* Report ECS-LFCS-91-180, Lab-

oratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1991

[Weg88] Peter Wegner. *The Object-Oriented Classification Paradigm.* In *Research Directions in Object-Oriented Programming* ed Bruce Shriver and Peter Wegner. The MIT Press 2nd edition 1988