MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

WORKING PAPER 180 FEBRUARY 1979

SECURITY AND MODULARITY IN MESSAGE PASSING

Carl Hewitt, Giuseppe Attardi, and Henry Lieberman

ABSTRACT

This paper addresses theoretical issues involved for the implementation of security and modularity in concurrent systems. It explicates the theory behind a mechanism for safely delegating messages to shared handlers in order to increase the modularity of concurrent systems. Our mechanism has the property that the actions caused by delegated messages are atomic. That is the handling of a message delegated by a client actor appears to be indivisible to other users of the actor. Our mechanism for delegating communications is a generalization suitable for use in concurrent systems of the subclass mechanism of SIMULA. Our mechanism has the benefit that it easily lends itself to the implementation of efficient flexible access control mechanisms in distributed systems. It is a generalization of the protection mechanisms provided by capability-based systems, access control lists, and the access control mechanisms provided by PDP-10 SIMULA.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

II -- INTRODUCTION

The implementation of a robust concurrent system requires very careful design. Not many conceptual or programming tools are provided to perform such a task. We address this problem by presenting some mechanisms to better structure a concurrent system. They are centered around a primitive for synchronization in message-based systems which is a further development of serializers [Atkinson and Hewitt 1978]. This primitive deals with problems of protection and modularity in the implementation of concurrent systems.

Protection is achieved by allowing message constructors to be given to different users. **Guardians** are abstractions that can implement the following functions for their resources: scheduling access, providing protection, and implementing recovery from hardware failures which manifest themselves as time-outs or data with an incorrect checksum. A guardian of protected resources will only perform tasks for messages which have been constructed by the appropriate message constructors. In a distributed system this constraint can be enforced using cryptography. Each of these messages understood by a guardian corresponds to an operation which the resource can perform. A message constructor for one of these messages can be communicated to those who are allowed to perform operation associated with the message. Serializers facilitate the implementation of efficient flexible access protection that subsumes the abilities of both capability based systems and access control systems.

Two important mechanisms to support modularity in our system are **delegation** and **inheritance**. Both of them are derived from a generalization of the subclass mechanism in SIMULA.

Delegation allows some of the requests received by an actor to be delegated to another actor called a handler. The delegated actor can be common to many actors thereby increasing the modularity of the system. Delegation is accomplished in such a way that handlers can perform atomic transactions for the actors which delegate requests to them.

Inheritance allows the sharing of descriptions common to many actors. Our description system facilitates the construction of a flexible hierarchy of descriptions. Invariant properties of actors can easily be expressed in this language which makes our description system useful for declaring the types of identifiers and state components in programs. The description system also is useful for expressing conditional tests to be performed in the course of the execution of programs.

The description language is used here for expressing properties for guardians (such as checking accounts) in distributed systems which communicate using message passing. A special language construct called a primitive serializer is presented which can efficiently implement guardians in a modular fashion. Furthermore primitive serializers provide mathematical behavioral denotations for the actors which they implement. We have developed a proof methodology for proving strong properties of network utilities e.g. a handler performs an atomic transaction for its client. This proof methodology is illustrated by proving properties of a handler which manages handles messages delegated by checking accounts. This research is a further step in a research program to develop a Programming Apprentice to aid software engineers in constructing and evolving large distributed software systems.

III -- PRIMITIVE SERIALIZERS

The design goals for monitors is that they were intended to be a structuring construct for implementing operating systems. There have been some attempts to develop useful proof rules for monitors [Howard: 1976; Gjessing: 1977; Hoare: 1974; Owicki: 1978] Serializers [Hewitt and Atkinson: 1976] are a further step toward these goals. However the language construct developed by Hewitt and Atkinson may be too complicated to be useful both as a formal foundation and as a basis for the proof methodology. In the study we present here the approach has been reversed. Instead of designing a desirable set of primitives and then trying to describe their semantics in a formal way, we started with a basic primitive with a simple semantics.

The general format of a simple primitive serializer is the following:

(*create_serialized_actor*
    [state:
        [component$_1$: description$_1$]
        ...
        [component$_n$: description$_n$]]
    [initialize:
        [component$_1$ ← expression$_1$]
        ...
        [component$_n$ ← expression$_n$]]
    [constraints: *...universally true properties declared here...*]
    [receivers:
        (pattern_for_message$_1$ *then* body$_1$)
        ...
        (pattern_for_message$_n$ *then* body$_n$)])

Serializers are introduced to create actors whose state may change after the receipt of a communication. A convenient way to express this is by means of the notion of state. The behavior of an actor depends on its (local) state, and its state may change as communications are received. The actors created by *create_serialized_actor* behave in the following way. A serializer can be either ***locked*** or ***unlocked***. When it is created it is ***unlocked***. When the first communication arrives, the serializer becomes ***locked***. If there are any receivers in the **receivers** section whose patterns match the message received, then the body of one of them is selected for execution.

When a result of the form

(*unlock* [reply: expression]
[component$_1$ ← expression$_1$]
...
[component$_n$ ← expression$_n$])

is computed then the serializer replies to the message it received with the value of expression and unlocks in a state with the state components component$_1$, ..., and component$_n$ having the values expression$_1$, ..., and expression$_n$ respectively.

Note that there are three separate events which must occur before a message M can be received by a serialized actor T. First it must be transmitted in a transmission event of the form

(*a* Transmission [target: T] [message: M])

Next it must arrive in a arrival event (synonymous with delivery event) of the form

(*an* Arrival [target: T] [message: M])

Hardware modules called arbiters are used to establish an arrival ordering for all communications delivered to T. Finally it must be received in a receipt event of the form

(*a* Receipt [recipient: T] [message: M])

Messages are received in the order in which they are delivered. A receipt event marks a transition in which the target changes from unlocked to locked. Thus if a serialized actor becomes locked then no more messages can be received until it unlocks.

IV -- DESCRIPTIONS of MESSAGES

As an example of how delegation is done using primitive serializers we give the implementation of a very simple checking account guardian.

There are two kinds of messages which must be dealt with by the guardian: **Withdrawal** and **Deposit** which can be described as follows:

```
(describe (a Withdrawal)
    [is:
        (a Message)
        (a Withdrawal [amount: (a Non_negative_US_currency)])])

(describe (a Deposit)
    [is:
        (a Message)
        (a Deposit [amount: (a Non_negative_US_currency)])])
```

which says that both kinds of messages have an attribute named *amount* which must be a non-negative US currency.

V -- DELEGATION of COMMUNICATIONS

Modularity is promoted by the SIMULA subclass mechanism by the way in which subclasses inherit procedures from their superclasses. Primitive serializers achieve this same effect in concurrent systems by facilitating the ability of an actor x to delegate communications which it receives to a handler which can be shared by a large number of actors that are similar to x. Usually the effects of a communication delegated by x should appear to be atomic to other users of x.

V.1 --- A Concurrent Case Expression

Clearly some kind of conditional test is needed in implementations. Use will be made of *select_case_for* expressions of the following form:

```
(select_case_for expression
   (pattern1 then body1)
   ...
   (patternn then bodyn)
   [none_of_the_above: alternative_body])
```

which when evaluated first evaluates expression to produce a value V.

If the value V matches any of the $pattern_i$ then the corresponding $body_i$ is evaluated and its value is the value of the *select_case_for* expression. If the value V matches more than one of the $pattern_i$ then an arbitrary one of the corresponding $body_i$ is selected to be executed. However, if the value of expression can match two different patterns then the Programming Apprentice will warn the user if it cannot demonstrate that the results of executing the bodies are indistinguishable. This rule has the advantage that it makes $body_i$ depend only on $pattern_i$ making it easy add more selections later.

We shall say that two activities are concurrent if it is possible for them to occur at the same. The concurrent case statement facilitates efficient implementation by allowing concurrent matching of expression against the patterns. This ability is important in applications where attempts to determine whether or not conditions hold take large amounts of time.

If the value V does not match any of the $pattern_i$ then alternative_body is executed. This rule provides the ability to have the patterns represent special cases leaving the alternative_body to deal with the general case if none of the special cases apply.

V.2 --- A Simple Guardian

Using these definitions we can implement the checking account guardian as follows by means of a serialized actor:

```
(define (create_account [initial_balance: =i (a Non_negative_US_currency)])
   [preconditions:]
   [is: (a Serialized_actor [responds_to: {(a Deposit) (a Withdrawal)}])]
   [definition:
      (create_serialized_actor
            [state: [balance: (a Non_negative_US_currency)]]
            [initialize: [balance ← i]]
            [receivers:
                ((a Withdrawal [amount: =a]) then
                   (select_case_for balance
                        ((≥ a) then
                          (unlock
                             [reply: (a Transaction_completed_report)]
                             [balance ← (balance - a)]))
                        ((< a) then
                          (unlock
                             [complaint: (a Transaction_not_completed [reason: overdraft])])))))
                ((a Deposit [amount: =d]) then
                   (unlock
                        [reply: (a Transaction_completed_report)]
                        [balance ← (balance + d)]))
                [delegate_to: account_handler]])])
```

V.3 --- A Simple Handler

If a message received by a serialized actor does not match any of the patterns of the receivers then then the alternative_receivers are applied. The alternative receivers will often be of the form

```
[delegate_to: handler]
```

which will result in handler being sent a message of the form

```
(a Buck_pass [message: m] [inside: inside_client] [outside: client])
```

where client is the serialized actor itself and inside_client is an unlocked representative of the client. This mechanism for delegating other communications is a slightly cleaned up and generalized version of the mechanism used in SIMULA [Birtwistle et. al.: 1973, Palme: 1973], SMALLTALK [Ingalls: 1978], and DIRECTOR [Kahn: 1978]. It was developed jointly with Ken Kahn.

Note that the above implementation of checking accounts delegates all communications which are not deposits or withdrawals to the actor account_handler which is defined below.

```
(describe (a Slip)
   [is:
      (a Slip
            [transactions:
               (a Sequence
                     [each_element:
                        (either
                           (a Deposit)
                           (a Withdrawal))])])])
```

If account_handler receives a communication which is a Slip with a sequences of transactions t, then it performs as many transactions as possible as an atomic operation and returns a list of the transactions that could not be done. The implementation below makes us of the unpack construct to manipulate sequences. The unpack construct is explained in an appendix. An account handler performs the transactions on a Slip for an account by going through the transactions in turn and attempt to perform each transaction. The account_handler communicates with inside_the_account in order to make its actions appear to be atomic to other external users.

We will use expressions of the form form (*send* t ← m) and (*send* m → t) to denote the result of sending the message m to the actor t.

In addition syntax of the *select_case_for* command is extended slightly to enable it to deal with complaints generated by the evaluation of expression so that its new syntax is:

```
(select_case_for expression
        (reply_pattern1 then reply_body1)
        ...
        (reply_patternn then reply_bodyn)
        [complaints:
           (complaint_pattern1 then complaint_body1)
           ...
           (complaint_patternj then complaint_bodyj)]
        [none_of_the_above: alternative_body])
```

The account_handler is unserialized (never changes state). This means that it can concurrently be handling messages delegated by several different accounts.

```
(define account_handler
   [is: (an Unserialized_actor [responds_to: (a Buck_pass [message: (a Slip)])])]
   [definition:
      (create_unserialized_actor
            [receivers:
                ((a Buck_pass [message: (a Slip [transactions: =t])] [inside: =inside_the_account]) then
                  (select_case_for t
                        ([] then (a Transaction_completed_report))
                        ([=first_t !=rest_t] then
                         (select_case_for (send inside_the_account ← first_t)
                              ((a Transaction_completed_report) then
                                  (send account_handler ←
                                          (a Buck_pass
                                            [message: (a Slip [transactions: rest_t])]
                                            [inside: inside_the_account])))
                              [complaints:
                                 ((a Transaction_not_completed_complaint) then
                                  (complain
                                     (a Transaction_not_completed_complaint
                                          [anomalous_transaction: first_t]
                                          [complaint: c]
                                          [remaining_transactions: rest_t])))])))])])
```

VI -- Atomicity of Transactions

We want to show that the handler for the account guardian processes a Slip containing a sequence of transactions as if they were a single atomic transaction. This means that no other request is considered by the guardian until the account handler has completed all its transactions. At this point some more explanations are needed of the semantics of delegation. Consider a serialized actor s which delegates a message m to a handler by an expression of the form:

```
[delegate_to: handler]
```

The actor s is locked by having received the message m. It will become unlocked when the handler will return with a value. The state of s will be the state as modified by the handler. The effect of delegating is a message of the form:

```
(a Buck_pass [message: m] [inside: inside_of_s] [outside: s])
```

being sent to the handler. The inside_of_s is a serialized actor similar to s with which it shares the state and the behavior. They have however distinct queues of incoming messages. If the inside_of_s happens to also delegate a message to a handler H, then it will send H a new actor representing the inside of inside_of_s. Each one of these representatives is conceptually a level of nesting and, at the same time, a level of locking. Each one can become unlocked while the actors of lower nesting level still remain locked. (One could compare this mechanism to the nesting of interrupts in some machines.) With this model in mind it is to prove that the transactions in a Slip are performed by the account handler as if they were a single transaction. In fact if a Slip is received by the account actor A, then A immediately locks. The slip is delegated to the account handler, and A will not become unlocked until the handler returns. A standard proof by induction on the length of the Slip proves that the handler always returns. When the handler makes a request to the account using

```
(send inside_the_account ← first_t)
```

inside_the_account remains locked while this request is processed.

VII -- PROTECTION

The use of abstraction allows us to protect actors from arbitrary manipulations which can destroy their integrity. A further level of protection is usually required to prevent the use of an operations by a particular class of users.

In our system, protection is achieved by allowing message constructors to be given or not to a user. A guardian of protected resources will only perform tasks for messages which have been constructed by the appropriate message constructors. In our previous example the message constructors are:

```
(a Withdrawal)
(a Deposit)
```

Each of these messages understood by a guardian corresponds to an operation which the resource can perform. A message constructor for one of these messages can be communicated to those who are allowed to perform the operation associated with the message. We can protect the account against withdrawals from unauthorized users by keeping them from knowing the constructor for withdrawal messages. In a distributed system this constraint can be enforced using cryptography.

Serializers facilitate the implementation of efficient flexible access protection that subsumes the abilities of both capability based systems and access control systems. For example, the set of users authorized to perform withdrawals can be kept distinct from the set of those authorized to perform deposits on the same account. An employee might want to give their employer the right to deposit money in their account but keep the right to make withdrawals to themselves!

VII.1 --- Protection using Classes

Protection mechanisms based only on data abstraction and scoping rules in general give the same access rights to all users which are within the scope of the protected resource. For example it is apparently impossible [Moller-Pedersen: 1978] to achieve the goal of having some users only be able to make deposits and others only to be able to make withdrawals from an account implemented using the class mechanism of SIMULA (even with the additions proposed by [Palme: 1973] that have been implemented on the PDP-10).

VII.2 --- Protection using Qualified Types

The incorporation of qualified types [Jones and Liskov: 1976] in programming languages has been proposed to permit the implementation of abstract data types in a way that enables compile time checking of access control. In order to investigate the issues, we provide an implementation in ACT1 of the associative memory example treated in their paper:

```
(define (create_associative_memory)
  [is: (a Serialized_actor [responds_to: {(an Insertion) (an Update) (a Lookup) (a Deletion)}])]
  [definition:
     (create_serialized_actor
          [state:
            [memory: (a Set [each_element: (an Entry)])]]
          [initialize:
            [memory ← {}]]


          [receivers:
            ((an Insertion [name: =n] [value: =v]) then
                (if (memory is {(an Entry [name: n]) ...})
                 then (unlock [complain: (an Insertion_error)])
                 else (unlock [memory ← (memory U {(an Entry [name: n] [value: v])})])))

            ((an Update [name: =n] [value: =v])
                (if (memory is {((an Entry [name: n]) which_is =e) ...})
                 then (unlock [memory ← ((memory - {e}) U {(an Entry [name: n] [value: v])})])
                 else (unlock [complaint: (an Update_error)])))

            ((a Lookup [name: =n])
                (if (memory is {(an Entry [name: n] [value: =v]) ...})
                 then (unlock [reply: v])
                 else (unlock [complaint: (a Lookup_error)])))

            ((a Deletion [name: =n])
                (if (memory is {((an Entry [name: n]) which_is =e) ...})
                 then (unlock [memory ← (memory - {e})])
                 else (unlock [complaint: (a Deletion_error)])))

            [delegate_to: associative_memory_handler]])])
```

Given an actor x, we can create a new actor which is restricted to handling a set of message types S using the following definition:

```
(define (restrict =x to =S)
  [definition:
    (create_unserialized_actor
        [receivers:
            (((a =message_constructor) which_is =the_message) then
                (if (message_constructor ∈ S)
                 then (send x ← the_message)
                 else (complain (an Access_control_error)))))]])
```

For example if v is an associate memory then

```
(restrict v to {Insertion Update})
```

will create a new actor which will only accept Insertion and Update messages. Intuitively, the restrictions on how an actor can be used are expressed by the restrictions imposed on the messages as they traverse a path to the actor. Thus, using one path rather than another to send messages to an actor changes the way the actor can be manipulated. For example, suppose v is an associative memory and

```
(x is (restrict v to {Lookup Insertion}))
(y is (restrict v to {Lookup}))
```

Using y it is impossible to modify v since only Lookup messages will reach v. On the other hand using x, Insertion messages can be sent to v which might cause v to change state.

VII.3 --- Protection using Trademarks

Trademarks [Morris: 1973] have been proposed as a very elegant basis for protection in programming languages. In this section we show how this proposal can be incorporated in the framework proposed here. We postulate the existence of a primitive procedure called **create_marker** which is called with no arguments an returns a newly created marker each time it is called. For each marker m, (m x) is an actor which behaves exactly like x in terms of its behavior when sent messages. Furthermore we postulate the existence of a primitive function called trade_mark such that for each marker m, (trade_mark m) is the trademark which is affixed by m. We require that for each marker m and actor x that

```
((m x) is (a Marked_actor [mark: (trademark m)]))
```

which says that the result of marking x using m bears the trademark of m. Furthermore we require that for any two markers m1 and m2 and any actor x,

((m1 (m2 x)) *is* (*a* Marked_actor [mark: (trademark m2)]))

which says that the order in which trademarks are applied is immaterial.

VIII -- FUTURE WORK

One important area in which work remains to be done is to demonstrate that primitive serializers can be implemented as efficiently as other synchronization schemes involving the use of semaphores, monitors, etc. We have designed primitive serializers with this goal in mind. On the basis of some preliminary investigation we believe that they can be implemented at least as efficiently as monitors and communicating sequential processes.

The third author has constructed some preliminary implementations in a dialect of the language described in this paper that runs on the PDP-10. In the course of the next year, we will continue to work to improve this implementation and to transfer it to the MIT CADR MACHINE where it can be supported by micro-code. In this way we will construct an efficient implementation in which no program depends on the physical representation of any actor. I.e. the behavior of any actor in the system can always be extended.

Another area in which work remains to be done is automating proofs such as the one in this paper. We feel that we are getting close to the point where a Programming Apprentice can do most of the proof under the guidance of expert programmers. Russ Atkinson is working on automating the proofs for the version of serializers in [Hewitt and Atkinson: 1977]. We hope to be able to use some of the techniques which he has developed in our symbolic evaluator.

IX -- CONCLUSIONS

In this paper we have developed a construct for delegating the implementation of concurrent operations on shared resources so that each operation appears to be atomic to other users of the resource. The construct generalizes the delegation mechanism used in SIMULA which consists of subclasses and virtual procedures. Apparently there is no convenient way to accomplish this delegation using monitors which has unfortunately led to the development of a confused literature on "nested monitor calls". Also there is apparently no way to accomplish delegation in Communicating Sequential Processes [Hoare: 1978]. Our proposal helps to make concrete some ideas in [Fisher: 1970] on "relative continuity".

The same mechanisms which support delegation also provide a protection mechanism that incorporates in distributed systems the most powerful and flexible capabilities of operating systems [Dennis: 1966, Lampson: 1971, Bishop: 1977] and programming languages [Palme: 1973, Morris: 1973, Jones and Liskov: 1976].

X -- ACKNOWLEDGEMENTS

This paper has benefited from ideas that sprang up in conversations in the summer and fall of 1978 with Jean Ramon Abrial, Ole-Johan Dahl, Edsger Dijkstra, David Fisher, Stein Gjessing, Tony Hoare, Jean Ichbiah, Gilles Kahn, Dave MacQueen, Robin Milner, Birger Moller-Pedersen, Kristen Nygaard, Jerry Schwarz, and Bob Tennent. The first author would like to thank Luigia Aiello and Gianfranco Prini and the participants in the summer school on Foundations of Artificial Intelligence and Computer Science in Pisa for helpful comments and constructive criticism.

Valdis Berzins, Howard Cannon, Dan Shapiro, Richard Stallman, Deepak Kapur, Vera Ketelboeter, and the members of the Message Passing Systems Seminar have given us valuable feedback and suggestions on this paper. Russ Atkinson is implementing a symbolic evaluator for the version of serializers in [Hewitt and Atkinson: 1977]. Vera Ketelboeter has independently developed a notion of "responsible agents" that is very close to the transaction managers described in this paper. Jerry Barber and Maria Simi have developed methods for proving the guaranteed response property for actor systems that allow the possibility of unbounded nondeterminism.

Although we have criticized certain aspects of monitors and communicating sequential processes in this paper, both proposals represent extremely important advances in the state of the art of developing more modular concurrent systems and both have deeply influenced our work.

The description system presented in this paper is intended as a first step toward the development of a **universal type system** in the sense that it is powerful enough to express essentially any invariant property of an actor. The intellectual roots of our description system go back to von Neumann-Bernays-Godel set theory [Godel: 1940], the ω-order quantificational calculus, and the lambda calculus. Its development has been influenced by the property lists of LISP, the pattern matching constructs in PLANNER-71 and its descendants QA-4, POPLER, CONNIVER, etc., the multiple descriptions and beta structures of MERLIN, the class mechanism of SIMULA, the frame theory of Minsky, the packagers of PLASMA, the stereotypes in [Hewitt: 1975], the tangled hierarchies of NETL, the attribute grammars of Knuth, the type system of CLU, the descriptive mechanisms of KRL-0, the conceptual representations of [Yonezawa: 1977], the class mechanism of SMALLTALK [Ingalls: 1978], the goblets of Knowledge Representation Semantics [Smith: 1978], the selector notation of BETA [Nygaard et. al.: 1977], the inheritance mechanism of OWL, the mathematical semantics of actors [Hewitt and Attardi: 1978], the type system in Edinburgh LCF, the XPRT system of Luc Steels, and the constraints in ThingLab. Conversations with Alan Borning, Scott Fahlman, William Martin, Allen Newell, Alan Perlis, Dana Scott, Brian Smith, and the participants in the "Message Passing Systems" seminar were extremely helpful in getting it nailed down.

PLASMA [Hewitt and Smith: 1975, Hewitt: 1977, Hewitt and Atkinson: 1977 and 1979, Yonezawa: 1977] adopted the ideas of pattern matching, message passing, and concurrency as the core of the language. It was developed in an attempt to synthesize a unified system that combined the message passing, pattern matching, and pattern directed invocation and retrieval in PLANNER [Hewitt: 1969; Sussman, Charniak, and Winograd: 1971; Hewitt: 1971], the modularity of SIMULA [Birtwistle et. al.: 1973, Palme: 1973], the message passing ideas of an early design for SMALLTALK [Kay: 1972], the functional data structures in the lambda calculus based programming languages, the concept of concurrent events from Petri Nets (although the actor notion of an event is rather different than Petri's), and the protection inherent in the capability based operating systems. The subclass concept originated in [Dahl and Nygaard: 1968] and adapted in [Ingalls: 1978] has provided useful ideas. However, as discussed in this paper, we have found it useful be able to separate the inheritance of attributes from the delegation of communications in concurrent systems. The subclass mechanism of SIMULA attempts to perform both the role of description system and the role of communication delegation.

The pattern matching implemented in PLASMA was developed partly to provide a convenient efficient method for an actor implemented in the language to bind the components of a message which it receives. This decision was based on experience using message passing for pattern directed invocation which originated in PLANNER [Hewitt: IJCAI-69] (implemented as MICRO-PLANNER by [Charniak, Sussman, and Winograd: 1971]). A related kind of simple pattern matching has also be used to select the components of messages by [Ingalls: 1978] in one of the later versions of SMALLTALK and by [Hoare: 1978] in a design for Communicating Sequential Processes. However both SMALLTALK

and CSP use assignment to pattern variables instead of binding which was used in PLANNER, SIMULA, and PLASMA.

XI -- BIBLIOGRAPHY

Birtwistle, G. M.; Dahl, O.; Myhrhaug, B.; and Nygaard, K. "SIMULA Begin" Auerbach. 1973.

Bishop, P. B. "Computer Systems with a Very Large Address Space and Garbage Collection". T-78. MIT Laboratory of Computer Science. May 1977.

Brinch Hansen, P. "The Programming Language Concurrent Pascal" IEEE Transactions on Software Engineering. June, 1975. pp 199-207.

Dennis, J. and van Horn E. C. "Programming for Multiprogrammed Computations. CACM 9, 3. pp 143-155. 1966.

Fisher, D. A. "Control Structures for Programming Languages" Doctoral Dissertation. Carnegie-Mellon University. 1970.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Specifying and Proving Properties of Guardians for Distributed Systems" MIT AI Lab Working Paper 172. December 1978. Revised February 1979.

Hewitt, C. "Concurrent Systems Need Both Sequences and Serializers" MIT AI Lab Working Paper 179. December 1978. Revised February 1979.

Hewitt, C. "Evolving Parallel Programs" MIT AI Lab Working Paper 164. December 1978. Revised January 1979.

Hoare, C. A. R. "Monitors: An Operating System Structuring Concept" CACM. October 1974.

Hoare, C. A. R. "Language Hierarchies and Interfaces" Lecture Notes in Computer Science No. 46. Springer, 1976. pp 242-265.

Hoare, C.A.R. "Communicating Sequential Processes" CACM, Vol 21, No. 8. August 1978. pp. 666-677.

Ingalls, D. H. H. "The Smalltalk-76 Programming System Design and Implementation" Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. January 23-25, 1978. Tucson, Arizona. pp. 9-16.

Jones, A. K. and Liskov, B. H. "A Language Extension for Controlling Access to Shared Data" 1976.

Lampson, B. W. "Protection" Proceedings of Fifth Annual Princeton Conference on Information Science ans Systems. Princeton. pp 437-443. 1971.

Moller-Pedersen, B. Private Communication. August, 1978.

Morris, J. H. "Protection in Programming Languages" CACM. January 1973. pp. 15-21.

Owicki, S. "Verifying concurrent Programs With Shared Data Classes" Formal Description of Programming Concepts edited by E. J. Neuhold. North Holland. 1978.

Palme, J. "Protected Program Modules in SIMULA-67" Swedish Research of National Defense. Report FOAP C8372-m3 (E5). July 1973.

Wulf, W. A.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; and Pollack, R. "HYDRA: The Kernel of a multiprocessor Operating System" CACM 17, 6. pp 337-345. 1974.

APPENDIX I --- Thumbnail Sketch of the Description System

This appendix presents a brief sketch of the syntax and semantics of our description system. A paper which more fully presents the description system and compares it with other formalisms which have been proposed is in preparation.

XI.1 --- Syntax

If <x> is a syntactic category then an expression of the form <x>* will be used to denote an arbitrary sequence of zero or more items separated by blanks in the syntactic category <x>. An expression of the form <x>† will be used to denote an arbitrary sequence of one or more items separated by blanks in the syntactic category <x>.

The following is the syntax for descriptions and statements:

<description> ::= <identifier> |
=<identifier> | *;the character = is used to mark local identifiers*
<statement> | *;note that statements (which are described below) are descriptions*
<attribute_description> |
<instance_description> |
<criterial_description> |
<mapping_description> |
<sequence_description> |
<set_description> |
<multiset_description> |
<instance_description> |
(<description> *viewed_as* <description>) |
(<description> *if* <statement>) |
(<description> *that_is* <description>) |
(<description> *such_that* <statement>) |
(∧ <description>†) |
(∨ <description>†) |
(*either* <description>†) |
¬<description> |
(<relation> <description>*)

<criterial_description> ::= (*the_only* <description>†)

<instance_description> ::= <indefinite_instance> | <definite_instance>
<indefinite_instance> ::= (<indefinite_article> <concept> <attribution>*)
<definite_instance> ::= (*the* <concept> <attribution>*)
;definite_instances are used only for criterial descriptions
<indefinite_article> ::= *a* | *an*
;there is no semantic significance attached to the choice of which article is used
<concept> ::= <description> *;note that this is ω order*

<attribution> ::= [<binary_relation_description>†: <description>†]
<binary_relation_description> ::= <description> *;note that this is ω order*

<attribute_description> ::= <projective_attribute_description> |
(<indefinite_article> <binary_relation_description> *of* <description>)
<projective_attribute_description> ::= (*the* <binary_relation_description> *of* <description>)
;expresses that <binary_relation_description> *is projective for* <description>
;see example below for an explanation of projective binary relations

<mapping_description> ::= [<description>†⟼ <description>†]

<sequence_description> ::= [<elements_description>*] |
<set_description> ::= {<elements_description>*} | *;{ and } are used to delimit sets*
<multiset_description> ::= {|<elements_description>*|} | *;{| and |} are used to delimit multisets*
<elements_description> ::= ... |
<description> |
!<description> *;! is the unpack construct*

<statement> ::= (<predicate> <description>*) |
<predication> |
(<description> *coref* <description>) | *;statement of coreference*
({<description>*} *each_is* <description>) |
(*and* <statement>†) |
(*or* <statement>†) |
(*xor* <statement>†) |
(*not* <statement>) |
(*implies* <statement> <statement>)

<predication> ::= (<subject> *is* <complement>)
<subject> ::= <description>
<complement> ::= <description>

Note that the syntax of our description system reads somewhat like template English [Hewitt: 1975, Bobrow and Winograd: 1977, Wilks: 1976] Thus for example we write (*an* Integer) in this paper instead of writing that (integer) as was done in PLANNER-71. However we also allow the use of instance descriptions such as (*the* Integer [>: 0] [<: 2]) to describe the Integer which is greater than 0 and less than 2.

We feel that it is quite important that a description expressed in template English correspond in a natural way with the intuitive English meaning. For this reason we use the indefinite article in attribute descriptions of such as the one below:

(4 *is* (*an* element *of* {2 4 6}))

where the binary relation element can occur multiply in an instance description such as the following:

(({2 4 6} *is* (*a* Set [element: 2] [element: 4])))

Attribute descriptions only make use of the definite article in cases like the one below

((*the* imaginary_part *of* (*a* Real)) *is* 0)

where the binary relation imaginary_part projectively selects the imaginary part of a Real. In this case the relation imaginary_part might be inherited from Complex via the following description:

((*a* Real) *is* (*a* Complex [imaginary_part: 0]))

For the purpose of describing mappings, I prefer the syntax

[=x⟼ ...x...]

[cf. Bourbaki: Book I, Chapter II, Section 3] to the syntax

(λ x. ...x...)

of the lambda calculus. For example the mapping cubes which takes a number to its cube can be described as follows:

(*describe* cubes
  [is: [=n⟼ n^3]])

XI.2 --- Axioms

The behavioral semantics of the description system is defined by its underlying message-passing semantics. The axiomatization given below is significant in that it represents a first attempt to axiomatize a description system of the power of the one described here. As far as I know previous to the development of this one, there did not exist similar axiomatizations for FRL, KRL, OWL, MDS, etc.

The most fundamental axiom is Transitivity of Predication which says that for any <description$_3$>

Transitivity of Predication

(*implies*
  (*and*
    (<description$_1$> *is* <description$_2$>)
    (<description$_2$> *is* <description$_3$>))
  (<description$_1$> *is* <description$_3$>))

Another fundamental axiom is Reflexivity of Predication which says that for any <description>

Reflexivity of Predication

(<description> *is* <description>)

Other important axioms are Commutativity, Deletion, and Merging:

Commutativity

((*a* <description$_1$> <attributions$_1$> <attribution$_2$> <attributions$_3$> <attribution$_4$> <attributions$_5$>) *is*
    (*a* <description$_1$> <attributions$_1$> <attribution$_4$> <attributions$_3$> <attribution$_2$> <attributions$_5$>))

which says that the order in which attributions of a concept are written is irrelevant,

Deletion

((*a* <description$_1$> <attributions$_1$> <attributions$_2$>) *is* (*a* <description$_1$> <attributions$_2$>))

which says that attributions of a concept can be deleted, and

Merging

(*implies*
  (*and*
    (<description$_1$> *is* (*a* <description$_2$> <attributions$_1$>))
    (<description$_1$> *is* (*a* <description$_2$> <attributions$_2$>)))
  (<description$_1$> *is* (*a* <description$_2$> <attributions$_1$> <attributions$_2$>)))

which says that attributions of the same concept can be merged.

Additional axioms are given below for other descriptive mechanisms:

Coreference

(<description$_1$> *coref* <description$_2$>) *if and only if*
(<description$_1$> *is* <description$_2$>) *and* (<description$_2$> *is* <description$_1$>)

Criteriality

(*implies*
(*and*
(<description$_1$> *is* (*the_only* <description$_3$>))
(<description$_2$> *is* (*the_only* <description$_3$>)))
(<description$_1$> *coref* <description$_2$>))

Definite Selection

((*the* <description$_1$> *of* (*a* <description$_2$> [<description$_1$>: <description$_3$>])) *is* <description$_3$>)

Indefinite Selection

(<description$_1$> *is* (*a* <description$_2$> [<description$_3$>: <description$_4$>])) *if and only if*
(<description$_4$> *is* (*a* <description$_3$> *of* (<description$_1$> *viewed_as* (*a* <description$_2$>))))

Constrained Description

(<description$_1$> *is* (<description$_2$> *such_that* <statement>)) *if_and_only_if*
(*and*
(<description$_1$> *is* <description$_2$>)
<statement>)

Qualified Description

(<description$_1$> *is* (<description$_2$> *that_is* <description$_3$>)) *if_and_only_if*
(*and*
(<description$_1$> *is* <description$_2$>)
(<description$_1$> *is* <description$_3$>))

View Point

((<description$_1$> *viewed_as* <description$_2$>) *is* (<description$_2$> *such_that* (<description$_1$> *is* <description$_2$>)))

Negation

(<description$_1$> *is* ¬<description$_2$>) *if_and_only_if*
(*not* (<description$_1$> *is* <description$_2$>))

Conjunction

(<description$_1$> *is* (∧ <description$_2$> <description$_3$>)) *if_and_only_if*
    (*and*
        (<description$_1$> *is* <description$_2$>)
        (<description$_1$> *is* <description$_3$>))

Inclusive Disjunction

(<description$_1$> *is* (∨ <description$_2$> <description$_3$>)) *if_and_only_if*
    (*or*
        (<description$_1$> *is* <description$_2$>)
        (<description$_1$> *is* <description$_3$>))

Exclusive Disjunction

(<description$_1$> *is* (*either* <description$_2$> <description$_3$>)) *if_and_only_if*
    (*xor*
        (<description$_1$> *is* <description$_2$>)
        (<description$_1$> *is* <description$_3$>))

Conditional Description

(<description$_1$> *is* (<description$_2$> *if* <statement>)) *if_and_only_if*
    (<statement> *implies* (<description$_1$> *is* <description$_2$>))

XI.3 --- Examples

XI.3.a --- Articulation

Additional axioms hold for each of the primitive descriptive mechanisms of the system. For example

(*describe* cubes
    [is: (*a* Mapping [=n⟼ n^3])])

can be articulated as follows:

(cubes *is* (*a* Mapping [1⟼ 1] [2⟼ 8] [3⟼ 27] [4⟼ 64] [5⟼ 125] ...))

where ... is ellipsis.

XI.3.b --- Sets and Multisets

Sets and multisets can be described in terms of mappings using the usual mathematical isomorphisms. For example

```
(describe {a b}
    [is: (a Mapping [a⟼ 1] [b⟼ 1] [¬a ∧ ¬b⟼ 0])])
```

describes the set {a b} as a mapping from a and b onto 1 since they are present in the set and everything else maps to 0 since there are no occurrences of other elements. Extending the same idea to multisets gives the following example:

```
(describe {|a b a|}
    [is: (a Mapping [a⟼ 2] [b⟼ 1] [¬a ∧ ¬b⟼ 0])])
```

XI.3.c --- Transitivity

If (3 *is* (*an* Integer [<: 4])) and (4 *is* (*an* Integer [<: 5])), it should be possible to conclude that (3 *is* (*an* Integer [<: 5])). This goal can be accomplished by the command

```
(describe <
   [is: (a Transitive_relation [for: Integer])])
```

which says that < is a transitive relation for Integer and by the command

```
(describe (a =concept [=R: (a =concept [=R: =m])])
   [preconditions: (=R is (a Transitive_relation [for: =concept]))]
   [is: (a concept [R: m])])
```

which says that if x is an instance of a concept which has a relationship R with something which is the same concept which has the the relationship R with m where R is a transitive relationship for concept, then x has the relationship R with m. This example of transitivity cannot be done in most type systems; the above solution makes use of the ω-order capabilities of our description system.

XI.3.d --- Projective Relations

If (z *is* (*a* Complex [real_part: (> 0)])) and (z *is* (*a* Complex [real_part: (*an* Integer)])) then by merging it follows that (z *is* (*a* Complex [real_part: (> 0)] [real_part: (*an* Integer)])) However in order to be able to conclude that (z *is* (*a* Complex [real_part: (> 0) (*an* Integer)])) some additional information is needed. One very general way to provide this information is by

```
(describe real_part
   [is: (a Projective_relation [concept: Complex])])
```

and by the command

```
(describe (a =C [=R: =description1] [=R: =description2])
   [preconditions:  (R is (a Projective_relation [concept: C]))]
   [is: (a C [R: description1 description2])])
```

The desired conclusion is reached by using the above description with C with bound to **Complex**, R bound to real_part, description1 bound to (> 0), and description2 bound to (*an* Integer).

XI.3.e --- Quantification and Existence

The treatment of local identifiers in our description system differs in an important respect from the treatment of universally quantified variables in naive ω-order logic where universal quantification implies existence. For example the following sentence clear holds in ω order logic:

$$\forall P\ \forall x\ (P\ x)\ \textit{if and only if}\ (P\ x)$$

From the above sentence the following follows by the usual rule for quantifiers:

$$\forall P\ \exists Q\ \forall x\ (Q\ x)\ \textit{if and only if}\ (P\ x)$$

Using the following definition for P

```
(define (P =x)
   [definition: (not (x x))])
```

we get

$$\exists Q\ \forall x\ (Q\ x)\ \textit{if and only if}\ (\textit{not}\ (x\ x))$$

Using ∃-elimination with Q_0 for Q we get

$$\forall x\ (Q_0\ x)\ \textit{if and only if}\ (\textit{not}\ (Q_0\ Q_0))$$

Substituting Q_0 for x we obtain Russell's paradoxical formula:

$$(Q_0\ Q_0)\ \textit{if and only if}\ (\textit{not}\ (Q_0\ Q_0))$$

However the above formula is a contradiction in our description system only if $(Q_0\ Q_0)$ is a Boolean which are described as follows:

```
(describe (a Boolean)
  [is: (either true false)])

(describe true
  [is:
     ¬false
     (a Boolean)])

(describe false
  [is:
     ¬true
     (a Boolean)])
```

We propose to restrict the rules of logic to statements which are Boolean. For example the rule of double negation elimination can be expressed as follows:

```
(describe (not (not =p))
   [precondition: (p is (a Boolean))]
   [is: p])
```

In this way we hope to avoid contradictions in our description system. In the course of the next year we will attempt to adapt one of the standard proofs to demonstrate its consistency.