CONGRESSUS NUMERANTIUM

VOLUME 85

DECEMBER, 1991

WINNIPEG, CANADA

A Specification Methodology to Support Automation of Office Procedures

Hossein Saiedian, Hassan Farhat and Mansour Zand
Dept. of Mathematics and Computer Science
University of Omaha, Omaha, NE 68182

It is often difficult and costly to develop systems that would automate office procedures. One reason for this difficulty is the unavailability of an appropriate office specification language that an office analyst could employ when specifying the expected functionality (or behavior) of the system. In this paper, we introduce an object-based language to formally specify the externally observable behavior of office objects as well as their computations. Object-based approach, as embodied in several languages, has proven to be a useful principle of software organization. Our proposed language is called ABSL and is based on Carl Hewitt's *actor* theory. The actor theory is chosen because not only it captures the abstract power of of object-orientation paradigm, but provides as well a mathematically precise abstract machine for analysis of asynchronous and concurrent computations. The central concepts in ABSL are *objects* and *message passing*. Every entity, whether abstract or concrete, that is relevant to some office computation is conceptually viewed and modeled as an object. Computations among the objects are uniformly modeled as patterns of message passing. An example is provided to show the expressiveness of specifications in ABSL.

General Terms: OFFICE AUTOMATION, SOFTWARE SPECIFICATION
Additional Key Words/Phrases: OBJECT-BASED TECHNIQUE, ACTOR MODEL

1 Introduction

Office automation is the body of knowledge that is concerned with the analysis, design, implementation and efficiency of the systems that transform information within an office. Office-specific systems for use in an office are often costly and difficult to implement and/or prototype. One reason for such difficulty is the unavailability of an appropriate specification methodology that an office analyst can use to formally describe the externally observable behavior of such systems at a high-level of abstraction. As a result, a great deal of effort, time and money is spent on "re-inventing" the wheel whenever a new office system concept is to be developed.

Computations in an office are very difficult to specify using the traditional techniques technique because office computations are diverse [3], inherently concurrent

[6], asynchronous [6, 19], distributed [10], nondeterministic [6], and event-driven [19]. Thus, to be effective, a specification methodology should be powerful and expressive enough to precisely and clearly express all aspects and relevant behavior of a system to be used in an office environment. We believe that if the support of an adequate specification methodology is provided, then the development of an office system can greatly be accelerated, many of the design difficulties can significantly be simplified and the design cost substantially reduced.

To address the above problems, we have developed a specification language for the formal specification of office procedures. This language is called ABSL (for Actor-Based Specification Language). As its name implies, ABSL is principally based on the formal theory of the actor model [5] with certain extensions. Our approach to designing a formal office specification language has emerged from research on models of concurrency and distributed computations. In addition to the actor model, we considered several other well-defined models as a semantic basis for an office specification language, including the Petri nets model [16], CSP [7] and CCS [13]. But we decided to take advantage of the fact that the actor model not only provides a mathematically precise abstract machine [1] for the analysis of concurrent, asynchronous and non-deterministic computations but captures as well the abstraction power of *object-oriented* modeling techniques.

2 Software Specification

The development of any large system has to be preceded by a specification of what is required. Without such a specification, the system's developers will have no firm statement of the needs of the would-be users of the system. The need for precise specification is accepted in most engineering disciplines. Computer systems are in no less need of precision than other engineering tasks.

Many aspects of a software system must be specified including its functionality, performance and cost. In this paper attention is focused on externally observable behavior of a system. In the traditional software development process, the specification phase, i.e., when the functions of software product are specified, plays a critical role. Its role is so critical that failure to properly carry out this phase is historically known to cause great financial losses. Hence software specification has been the subject of great deal of attention in recent years. The existence of workshops and conferences devoted to this subject, e.g., the *International Workshop on Software Specification and Design* series, bears witness to the importance of this field [12].

The term specification refers to both *product* and *process*. As a process, the specification is the phase where the functionality of a system is defined. This process includes both specification generation and specification validation.[1] As a product, a specification is the document in which expected behavior and functionality of a systems is recorded. As a product, a specification play two key roles: first, it is

[1]The purpose of specification validation is to check whether a specification which has been derived is consistent with, and complete with respect to users' intent.

a contract between the users and the programmers; second it is a primary design document for the programmer.

To summarize, a specification is a piece of text that describes the behavior a software system. Such a specification should be much shorter than a corresponding implementation[2] and may serve for the following purposes [8]:

- Specifications describe the abstractions made and thus serve as system documentation.
- Specifications may serve as a mechanism for generating questions about functionality of the system and its intrinsic properties.
- A specification document can serve as a contract between the designers of a program and its customers and in essence binds the customers and designers by expressing the conditions under which the services of a program are legitimate and defining the results when these services are called.
- With respect to program validation, specifications may be very helpful to collect test cases.

There is a general agreement that in large software projects, *appropriate* specifications are a must in order to obtain quality software. Informal specifications alone are certainly not appropriate because they are incomplete, inconsistent, inaccurate, ambiguous and they rapidly become bulky. (For an excellent discussion of drawbacks of informal specifications, the readers are invited to read [11].) Inadequate specifications of software systems has long been quoted as a root cause of software failure and very high software maintenance cost. With a software being used in more and more critical office applications, it becomes increasingly vital to find ways of reducing the ambiguity and inconsistency in the specifications.

A formal specification methodology is comprised of a relatively small vocabulary of keywords and operators with a well-defined syntax and semantics. A formal specification methodology is a valuable tool in systems development since it permits a systems analyst to describe the external behavior of a proposed system precisely without specifying issues related to implementation. Another potential uses of a specification methodology is that it serves as a communication tool between the systems analyst and the users and implementors since it enables the analyst to describe the proposed system more precisely and unambiguously to them. Although specification methodologies have been applied to a wide area of software systems [4], there has been little work in developing a methodology to address the problem of specifications of office entities and computations.

The goal of this paper is to introduce a particular approach to specifications of office procedures. This approach is called ABSL and is based on the formal theory of the actor model. The rest of this paper is as follows: Section 3 briefly reviews the actor model and provides reason why this model was selected. In Section 4, an

[2]The key brevity is abstraction: the specification of a system should abstract away issues which relate to implementation.

informal description of objects in ABSL is given. Section 5 provides a more formal description of ABSL' objects. Section 6 includes the description of certain properties of the ABSL methodology. Section 7 includes an example to show expressiveness of ABSL specifications. In Section 8, we conclude this paper and discuss areas for further research.

3 Selection of Actor Model

The actor model [5, 2, 1] is a model of distributed computing in which every computational entity is represented as an autonomous object or an *actor*. In this model, no distinction is made between between procedure and data; both are represented as actors. The only means by which an actor may affect the behavior of another actor is by sending a message to it. All computations in a system of actors are thus represented by means of message transmission. A message transmission is referred to as an *event*. Thus the only things that happen in an actor system are events. An event marks the arrival of a message at the message target. The basic concepts of the actor model are simple, yet very general and powerful to express essentially any kind of computations. The motivations for using the actor theory was to explore its usefulness and to use its simple concepts as a basis for methodology.

The actor model has traditionally been used as a model of computation in the area of artificial intelligence. This model, however, has a number of characteristics that relate well to the computations in an office environment:

1. Computations of the actor model are *event-driven* where an event is defined as the arrival of a message. Events in the actor model are the only source of energy for computations [1]. This concept (i.e., event-driven computations) also resembles the activities of an office environment. The activities in an office environment are event-driven. An event in this context may be viewed as the arrival of a message (e.g., an admission form) into an office. The arrival of each such message may lead to creation of additional messages (e.g., copies of admission form sent to the graduate committee members) and thus creating and continuing additional activities.

2. The computations of the actor model imply an overwhelming amount of parallelism [1]. Similarly, office computations are highly concurrent [6].

3. The communications in the actor model are asynchronous. Likewise, the communications in an office are asynchronous. Furthermore, the communications in the actor model take place by means of message passing. The message passing paradigm of the actor model also resemble the communication patterns in an office.

4. The laws of the actor model ensure the *liveness* property in a distributed computing environment. The basic idea of the liveness property is to show that those events that are expected to occur eventually will occur. As observed in

[17], an important property in office systems is the liveness property or the guarantee of service.

5. The actor model supports an object-oriented style of computation. The concepts of an object and of object-oriented computations emerged in 1960's and are applied in diverse areas of computer science for the purpose of managing the complexity of software systems. The object-orientation is one of the most active area of research [18, 9]. Object-oriented techniques are particularly apt to deal with office applications [15, 14].

6. The actor model has a well defined mathematical basis [1] and provides a sound foundation for a partial ordering theory of distributed computations. As mentioned earlier, office computations are distributed.

In the actor model, the only means by which an actor may affect the behavior of another actor is by sending a message to it. All computations in the actor model are thus represented as message transmission. A message transmission is refereed to as an *event*. Thus the only things that happen in an actor system are events. An event marks the arrival of a message. Events are ordered in two ways: messages arriving at a given actor have a unique linear order called the ***arrival order*** while an event causing other events precedes those events in the ***activation order***. The transitive closure of these two ordering provides a partial order of events called the ***combined order*** [5]. The laws associated with the combined ordering include:

- No event precedes itself in the combined ordering. (This law is referred to as Strict Causality Law.)
- A chain of events in the activation ordering from an event E_1 to event E_2 is finite. (This law is referred to as Finite Activation Law.)
- The set of immediate predecessor of an event E is finite. (This law is called Finite Immediate Activation Successor Law.)
- If event E_1 precedes event E_2, then there are only finitely many events between E_1 and E_2. (This is referred to as Finite Intermediate Chain in the Combined Ordering Law.)
- Each event has finitely many immediate successors and finitely many immediate predecessor.
- If two events, E_1 and E_2, occur at the same target (i.e., if two messages arrive at the same actor), then either E_1 occurs before E_2 or E_2 occurs before E_1. In other words, two messages cannot be accepted at the same time. (This law is called Total Arrival Ordering Law.)
- No event can cause infinitely many events.
- The set of actors (objects) created by an event is finite. (This is called the Finite Creation Law.)

- Each event is generated by the sending of at most a single message.

The purpose of the actor laws is to restrict a system of actors to those computations that are physically realizable and to ensure that an actor system can physically be implemented. For a complete description, analysis and implication of actor laws see [5].

4 Object Representation in ABSL

ABSL is intended to serve as a tool to conceptually describe an office system in the framework of object-based approach. ABSL supports an *object-based* approach to the specification of office computations. In this approach, virtually every entity, whether abstract or concrete, that appears in an office is uniformly viewed as an object that can send or receive messages.

Each object consists of a local memory (or environment) containing tightly coupled data that are used to define the *state* of that object as well as all the operations that act on the local data. These operations are referred to as *behavior rules*. Thus an object consists of

- a local memory definition called *environment definition* defining features and important properties of an object, and

- a *behavior definition* that defines the actions or functions performed by the object when it receives a message.

In particular, the behavior definition (analogous to actor's *script*) specifies the type of messages accepted, operations performed on the local data, the new objects created, and/or the messages sent in response to the incoming messages. The behavior definition consists of a set mutually independent behavior rules. The overall structure of objects in ABSL is shown in Figure 1 while the general format of a behavior rule is shown in Figure 2. (ABSL keywords are `typed`.)

The `requires` and `effects` clauses in Figure 2 are used to capture the state of an object before and immediately after processing a message. ABSL supports a first order predicate calculus sublanguage to express the state-before and state-after of an object. This sublangaueg expresses the states of an object in terms of pre- and post-condition expressions where

- Pre-conditions are assertions about the state of an object before it can accept a certain message, and

- Post-conditions are assertions about state of an object that will be obtained or will prevail after the processing completion of an accepted message.

ABSL has several distinctive characteristics. First, it allows one to specify the behavior of office entities independent of their physical representation in an abstract and non-procedural fashion. Since all computations are represented as patterns of

```
object object-name --Object header
        with clause  --List of parameters this object may be initialized with
        copy clause  --Data types to be imported to this object;
        state-def  --Object environment definition
              - state variables are declared here
        end-state
        behavior-def  --Behavior definition of an object
              behavior-rule 1
              behavior-rule 2
                  ⋮
              behavior-rule n
        end-behavior
end object-name
```

Figure 1: The Overall Structure of an Object in ABSL

```
accept [message-tag(message parameters) delegate-to: d-path, sender: sender-name]
        requires: set of pre-conditions that must be true
                send-to statement for asynchronous communication
                send-wait statement for synchronous communication
                reply statement to reply to the message sender
                new statement to create new objects
        effects: set of post-conditions that are true after processing a message
end-accept
```

Figure 2: The General Format of a Behavior-Rule

message passing among the objects, the only things that are important to the users (and the implementors) about an object is what kind of messages that object may receive and what it does in response to the accepted messages. No attempt is made to describe how an object perform its actions; only the kind of messages it accepts. Such descriptions are abstract and independent of any implementation bias.

Second, since office applications constantly evolve, one of our main goals in developing ABSL has been to facilitate one with a notation to conveniently develop partial description of behavior of an object which describes the known properties of that object and to further specify additional features (*i.e.*, behavior rules) of that object in an incremental fashion. This goal becomes specially important if one considers the facts that

- it is usually difficult and often impossible to arrive at a complete specification for complex systems and

- properties of objects in an office are not available all at once; they often evolve with time.

Third, since ABSL specifications are structured as a collection of objects with well-defined interfaces, they can directly be mapped into an object-oriented implementation via an object-oriented language. Object-oriented programming is one of most active areas of research in computer science [18] and interest in this area continues to increase.

5 Formal Definition of an ABSL Object

The definition of an ABSL object is given as follows:

Definition 1 *Object — An object θ is a 2-tuple (ξ, β) where*

- *ξ is the local environment of object θ (ξ in particular is the union of Σ and $\overline{\Sigma}$, object θ's states before and after processing a message), and*
- *β is object θ's behavior.*

□

The local environment of object θ defines the *state* of θ at various time references. Object θ's behavior, on the other hand, defines the set of messages the θ accepts and its response to the accepted messages. Formally, the behavior of an object θ is a function that maps the incoming messages to a set of created objects, a set of communications sent to other objects, and the new state of object θ which results from processing the accepted message:

Definition 2 *Behavior — An object θ's behavior, β, is a function*

$$\beta : \Sigma^{\theta} \times \Gamma \rightarrow (\Theta \times \gamma\prime) \times \overline{\Sigma}^{\theta}$$

where

- *Σ^{θ} represents the state of object θ before the arrival of message $\gamma \in \Gamma$;*
- *Γ is a universal set of messages that object θ can accept;*
- *Θ represents the finite set of objects created as a result of processing message γ;*
- *$\gamma\prime$ represents the communications sent to other objects in respond to message γ and,*
- *$\overline{\Sigma}^{\theta}$ represents the state of objects θ after processing γ.*

□

The above definition implies that an object θ may exhibit a new state after processing a message γ, which may not be the same as its state before it accepting γ. An object which may change state is said to be *serialized*. The state of *unserialized* objects remains the same regardless of the number of messages processed:

Definition 3 *Unserialized Object — An object θ with behavior β and state Σ is an unserialized object if*

$$\forall \gamma \in \Gamma, \Sigma^{\theta} \times \gamma \rightarrow (\Theta \times \gamma\prime) \times \overline{\Sigma}^{\theta} \Rightarrow \Sigma^{\theta} = \overline{\Sigma}^{\theta}.$$

□

Unserialized objects are those objects that have no state. In other words, they are memory less and thus they can be used to represent mathematical functions.

The relationship between state-before (Σ) and state-after ($\overline{\Sigma}$) can be viewed as an object's *state transition*. Suppose an object has only one behavior rule (i.e., it accepts only one message). The relationship between the state-before and state-after of an object is defined as:

Definition 4 *Object State Transition — Let ϕ and ϕ' denote the set of assertions in* `requires` *and* `effects` *clauses of an object respectively. Then*

$$< \phi > \rightarrow O \rightarrow < \phi' >$$

is regarded as if an object is activated in a state for which ϕ holds, the object will perform operations in O and will be in a state for which ϕ' will hold.

□

In other words, ϕ is intended to describe exactly those states from which processing of a message is guaranteed to establish ϕ'. The above definition can be extended for objects with more than one behavior rule by simply considering the union (or conjunction) of both pre- and post-conditions of each behavior rule.

6 History of Computations

Acceptance of a message by an object is called an event. The behavior involving only a single object is a totally (linearly) ordered sequence of events. Thus the history of computations of an object can be viewed as a sequence of events. Events in this history are distinguished by their place in the total ordering of events. The state of an object can be captured by recording the events in which it participates. Given a history of computations for an object, one can determine whether the object has participated in a particular event or not. Suppose S represents a sequence of events that an object has participated in:

Definition 5 *Sequence — S is a sequence of events occurring at an object 0 if*

$$S = < e_1, e_2, ..., e_n > \text{ and } e_1 \succ e_2 \succ ... \succ e_n$$

where $\succ$ means "arrived before".

□

Using this definition, one can determine whether a given event e_i belongs to the computation history of an object *0*:

Theorem 1 *Suppose sequence S represents the computational history of object 0. Function $\psi(e, S)$, defined below, evaluates to true if a given event e belongs to the history of object 0, false otherwise:*

$$\psi(e, S) = \begin{cases} true & \text{if } e = car(S) \vee \psi(e, cdr(S)) = true \\ false & otherwise \end{cases}$$

where

- $\psi(e, <>) = false$,
- $car(< e_1, e_2, ..., e_n >) = e_1$, *and*
- $cdr(< e_1, e_2, ..., e_n >) =< e_2, ..., e_n >$.

Proof: According to the Total Arrival Ordering *law,*

$$\forall e_i, e_j | target\_of(e_i) = target\_of(e_j) \Rightarrow e_i \succ e_j \vee e_j \succ e_i$$

where target_of(e_i) means "the target object where event e_i occurred at" and $\Rightarrow$ means "implies". (The above law says that all events occurring at an object are linearly ordered.)

The Finitely Many Predecessor in Arrival Ordering *law states that*

$$\forall e_i, e_j | target\_of(e_i) = target\_of(e_j) \Rightarrow \{e_k | e_i \succ e_k \succ e_j\} \text{ is finite}$$

Technically speaking, the above law implies that the arrival ordering forms a finite descending chain such that the process of repeatedly taking the predecessor of an event in the chain will eventually terminate. These two laws imply that the theorem can be proved inductively on size of S, shown as $|S|$:

Suppose $|S| = 0$. Thus $S =<>$ and by definition, $\psi(e, S)$ =false since $e \notin S$.

Induction Hypothesis: Assume for some $n > 0$ that if $|S| = n$, then $\psi(e, S) = true$ if $e \in S$.

Suppose $|S| = n + 1$. There are two cases to consider:

- *Case 1: $S = < e, \#S >$ for some sub-sequence $\#S$. Thus $e = car(S)$ and by hypothesis, $\psi(e, S) =$true since $e \in S$.*

- *Case 2: $S = < e', \#S >$ and $e \neq e'$. Then by definition, $\psi(e, S) = \psi(e, cdr(S))$. Since $|cdr(S)| = |\#S|$ and by induction hypothesis, $\psi(e', cdr(S)) =$ true, and $e \in cdr(S)$. Therefore, $\psi(e, S) =$ true if $e \in cdr(S)$ which is true if $e \in S$.*

□

The behavior of an object can be described by specifying the set of operations that it performs once it receives a message. Furthermore, given the past history of events an object has participated in, one can determine the reaction (response) of that object to the next event. This implies that the set of past messages that an object θ has accepted must be recorded to predict its future behavior. For practical purposes, however, it may not be desirable to keep such a record. Alternatively, one can apply the concept of *state* from the sequential systems to define an object's response to incoming messages. In the context of sequential systems, the concept of state allows the future behavior of the system to be completely predicted by *abstract* state of the system at the time of an event instead of the whole past history of the system [2]. (Baker [2] more formally referred to the abstract state of the system as an *equivalence class* of the past histories of the system.) The above concept of abstract state can be used to define the state of individual objects: the state of an object will be defined by the abstract state of that object at the time of a message arrival. terms of the equivalence class of its past histories (*i.e.*, the past messages accepted). Thus if an object θ is in state Σ, then θ can accept a message γ and exhibit a new state shown as $\overline{\Sigma}$. ABSL supports a first order predicate logic sub-language to express Σ and $\overline{\Sigma}$, (the state of object θ before and after processing a message). The predicates of this language express the state of an object in terms of pre- and post-conditions (or I/O predicates) where pre-conditions are assertions about state (Σ) of an object before accepting a message and post-conditions are assertions about state ($\overline{\Sigma}$) of an object that will be obtained or will prevail after the processing completion of γ.

As discussed previously, each object has a well-defined local time which linearly orders the events as they occur at that object. These local orderings can be related to each other by the *activation ordering*. The activation ordering specifies the causal relationship between events happenings at different objects. The causal relationship is a partial ordering relationship in which events occurring at different objects are unordered unless they are connected by direct or indirect causal links. Thus, the computational history of a system of objects is a partially ordered set of events obtained by the transitive closure of *activate* and *arrival* ordering relations. The partial ordering relation approach to semantics of computations is well suited to specify the behavior of objects in a distributed system: Instead of recording the current state of system, changes in state are reflected in change over time in behavior of individual objects. In other words, since the causal relation ordering among the events in a distributed environment does not specify a unique total orderings of events [1], the notion of computations are generalized from a total ordering of events to a partial orderings of events. As a result, two events are ordered only if they are related. In other words, as long as the changes made to an object at a given time do not affect other objects, the state of those objects need not be considered when an object receives a message. That is, one needs only to consider the effects of an event e on the local state of object 0 and those events that will be generated once 0 participates in event e. The semantics of computations is defined as the effects of a message on the local state of its recipient.

It is also important to show that all the expected computations in system of ABSL will take place. Since computations are expressed as patterns of message passing, we must show that all the messages sent by various ABSL objects will eventually be received by the expected recipients. This is shown by the following theorem.

Theorem 2 *All pending messages will eventually be processed. Stated otherwise, all expected events will eventually occur. This theorem essentially expresses the* liveness *property of message passing in* ABSL*: each message sent by an object will "eventually" be received by its expected recipient.*[3]

Proof. Let $n \in$ N. (N is the set of natural numbers). Let $\{0_i\}_{i=1}^{n}$ be a set of objects such that $i \neq j \Rightarrow 0_i \neq 0_j$. Thus the set of objects in an environment is finite. This is implied by the Finite Creation *law.*

According to the Finite Immediate Activation Successor *law, each object may send a* finite *number of messages to other objects after receiving a message. Suppose each object 0_i sends γ_j messages where $\{\gamma_j\}_{j=1}^{m}$ are the messages and $m \in$ N. Let M be the total number of messages sent by object 0_i. Thus*

$$M = \sum_{i=1}^{n} \sum_{j=1}^{m} 1 < \infty$$

In other words, the total number of pending messages at any given time is finite. Choose an arbitrary $k, 1 \leq k \leq n$. Let m' be the sum of messages sent to 0_k. Thus $m' \leq M$. The number of messages arriving at object 0_i is therefore finite. As a result, object 0_i will receive and process each message sent to it after some time.

Other properties of ABSL objects and their computations (e.g., possibility of deadlock occurrence, *configuration* transition, synchronization, etc.) can be found in [17].

7 An Example: A Doctor's Office

In this section, we provide a simple example to show the expressiveness of specifications in ABSL. In the example, the entities of a simple health clinic office are represented as objects. (Due to lack of space, this example is greatly simplified.) There are two doctors and one receptionist, each modeled as objects. There is also a patient-queue object that queues the patients in a FIFO discipline. The patient objects walk to the clinic and send a `Patient-In` message synchronously to the receptionist. After receiving this message, the receptionist may do one of the following:

[3]Note: One important assumptions is made here: the underlying communication network is *reliable*. This assumption is an implementation-dependent consideration.

- Ask the patient object to see doctor #1, if doctor #1 is not busy,
- Ask the patient object to see doctor #2, if doctor #2 is not busy,
- Ask the patient to wait; the patient's name is sent to the patient-queue object to be queued.

(The messages from the patient objects are sent synchronously because each patient needs to know immediately what to do after entering into the clinic). Thus the receptionist object needs to reply to them as soon as possible and as a result, the task of queuing/dequeuing patient objects is given to the patient-queue object. Doctor objects interact with the patient-queue object. Once a doctor object is ready to see a patient, it sends a message (`Doc1-Ready` or `Doc2-Ready`) to the patient-queue object. If the patient-queue object has some patient objects queued, it sends a message (`See-Doc1` or `See-Doc2`) to the object that has been waiting for a longer time and asks it to see the doctor. If there are no patient objects queued, then the patient-queue object sends an appropriate message (`Doc1-Available` or `Doc2-Available`) to the receptionist object. The receptionist object thus receives messages of the form `Doc1-Available` or `Doc2-Available` if one of the doctors has effects available. After receiving such a message, the receptionist object changes its state appropriately. It is assumed that the patient objects have the courtesy to let the receptionist know if they decide to leave the clinic before seeing a doctor. Thus, if the receptionist object receives a message `Patient-Out` from a patient, it sends a `Remove` message to the patient-queue instructing it to remove the corresponding patient object from the queue. A pictorial representation of the doctors' clinic is given in Figure 3.

The specification of the receptionist object is given in Figure 4.[4] The receptionist object only knows the patient-queue object but not the names of the patient object since there may be many patients coming to the clinic and thus the receptionist is not required to know their names.

The specification of the patient-queue object is straightforward. It accepts messages of the form `Enqueue, Doc1-Ready, Doc2-Ready` or `Remove`. (See Figure 5) Patient objects are queued in a first-in first-out basis. The specification of the patient-object is not given due to lack of space.

8 Conclusions and Further Research

In this paper, we proposed a methodology called ABSL as a tool for specifying applications for office automation. ABSL is a non-procedural specification language. Its key distinguishing aspects are its theoretical foundation (i.e., the actor theory) and its

[4]Comments about the notation: The unpack operator (shown as #) denotes a (possibly empty) subsequence. Thus, the sequence $< e$ #S$>$ consists of an element e and a subsequence #S. An identifier inside the **effects** clause is adorned with a prime symbol (') to denote its new value after a message has been processed. Input parameters to a behavior-rule are decorated with a "!" symbol (e.g., name!) while output parameters are decorated with a "?" (e.g., response?). As shown in the first example, a message package may lack any parameter. In such a context, a message serves merely as a signal.

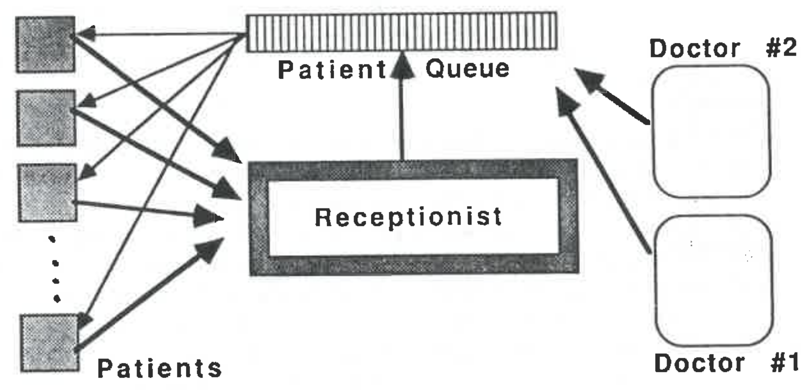

Figure 3: Pictorial Representation of Objects in Doctors' Office

uniform representation (office entities are represented as objects while computations are presented as patterns of message passing).

The long-term purpose of a specification methodology is not the static description of a system, but also its dynamic derivation. The design of ABSL is thus not pragmatically complete without a methodology for its systematic use. A specification language, in effect, defines a nondeterministic solution space, i.e., a nondeterministic algorithm for deriving a solution to a problem within a given domain. A solution is computationally infeasible unless the dimensionality of a solution space is reduced. Methodological heuristics are the way this dimensionality is handled in practice. Should a methodology ever reduce the dimensionality of a solution space to its minimum, that is one, so that the solution space becomes deterministic, then the entire derivation process can be automated. An interesting area of research would be to perform major methodological research using ABSL as a base, develop automated tools for this analysis and determine whether the methodology is deterministic.

Most object-oriented programming languages support code re-use in the construction of families of related components using the concept of *inheritance* Inheritance allows a component to be defined in terms of one or more other components, inheriting attributes from those components and overriding them when necessary. In essence, the concept of inheritance provides a mechanism for information sharing.

In this work, the concept of inheritance has been ignored since it is not integral to the actor model and thus no specific scheme for inheritance is inherent to the actor model [1]. This is primarily because the actor model does not support the concepts of object *class* hierarchy of classes and *meta* classes. It would be interesting to investigate

```
object receptionist
state-def
        doc1-free, doc2-free: boolean initially true
        patient-queue: object
end-state
behavior-def
        event [Patient-In(patient!: object, response?: str)]
              case requires: doc1-free = true
                          response? = "Doctor 1 Available" then reply-to: patient!
                    effects: doc1-free' = false ∧ doc2-free' = doc2-free
              case requires: doc2-free = true
                          response? = "Doctor 2 Available" then reply-to: patient!
                    effects: doc2-free' = false ∧ doc1-free' = doc1-free
              case requires: doc1-free = false ∧ doc2-free = false
                          response? = "Please Wait" then reply-to: patient!
                          send-to: patient-queue [Enqueue(patient!: object)]
                    effects: doc1-free' = false ∧ doc2-free' = false
              end-case
        end-event
        event [Doc1-Available()]
              requires: doc1-free = false
              effects: doc1-free' = true
        end-event
        event [Doc2-Available()]
              requires: doc2-free = false
              effects: doc2-free' = true
        end-event
        event [Patient-Out(patient!: object)]
              send-to: patient-queue[Remove(patient!: object)]
        end-event
end-behavior
end receptionist
```

Figure 4: Specification of Receptionist Object

```
object patient-queue
state-def
      wait-list: sequence of object initially < >
      receptionist: object
end-state
behavior-def
      event [Enqueue(patient!: object)]
           requires: wait-list = <#waiting>
           effects: wait-list' = <#waiting patient!>
      end-event
      event [Doc1-Ready()]
           case requires: wait-list = <p #waiting>
                     send-to: p [See-Doctor1()]
                effects: wait-list' = <#waiting>
           case requires: wait-list = < >
                     send-to: receptionist [Doc1-Available()]
                effects: wait-list' = < >
           end-case
      end-event
      event [Doc2-Ready()]
           case requires: wait-list = <p #waiting>
                     send-to: p [See-Doctor2()]
                effects: wait-list' = <#waiting>
           case requires: wait-list = < >
                     send-to: receptionist [Doc2-Available()]
                effects: wait-list' = < >
           end-case
      end-event
      event [Remove(patient!: object)]
           requires: wait-list = <#waiting1 patient! #waiting2>
           effects: wait-list' = <#waiting1 #waiting2>
      end-event
end-behavior
end patient-queue
```

Figure 5: Specification of Patient-queue Object

and see what role inheritance can play in ABSL methodology and whether the notion of inheritance would provide a substantial leverage for the analysis and specifications of office systems.[5]

References

[1] Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[2] Baker, H.: *Actor Systems for Real-Time Computations*, PhD Dissertation, MIT, Cambridge, Mass., 1978.

[3] Bracchi, G. and Pernici, B.: "The Design Requirements of Office Systems," *ACM Trans. on Office Information Systems*, Vol. 2, No. 2, April 1984.

[4] Gehani, N. and McGettrick, A.D. (Editors): *Software Specification Techniques*, Addison-Wesley, 1986.

[5] Hewitt, C. and Baker, H.: "Actors and Continuous Functionals," in *Formal Description of Programming Concepts*, E. Neuhold (editor), North-Holland, 1977.

[6] Hewitt, C.: "Offices Are Open Systems," *ACM Trans. on Office Information Systems*, Vol. 4(3) (July 1986), pp. 271-287.

[7] Hoare, C.A.R.: "Communicating Sequential Processes," Communications of ACM, Vol. 21, No. 8, 1978.

[8] Horebeek, I.: *Algebraic Specification in Software Engineering*, Springer-Verlag, 1989.

[9] Kim, W., and Lochovsky, F.H. (editors): *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, 1989.

[10] McBride, R.A. and Unger E.A.: "Modeling Jobs in a Distributed System," in *Proc. of ACM Symp. on Small and Personal Computers*, 1983.

[11] Meyer, B.: "On Formalism in Specifications," *IEEE Software*, Jan. 1985, pp. 6-26.

[12] Mili, A., Boudriga, N., and Mili, F.: *Towards Structured Specifying*, Halsted Press, 1989.

[13] Milner, R.: *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag, N.Y., 1980.

[14] Nierstrasz, O.: "Integrated Office Systems," in [9].

[5]Incorporated into ABSL is a limited version of information sharing through the use of copy clause in an object header. The use of copy clause is restricted to the sharing of data types.

[15] Pernici, B.: "Objects with Roles," in *Proc. of 1990 ACM Conf. on Office Information systems*, pp. 205-215, 1990.

[16] Peterson, J.L.: "Petri nets," *Computing Surveys*, Vol. 9, No. 3, Sept. 1977.

[17] Saiedian, H.: *An Object-Oriented Approach to the Specification of Applications for Office Automation*, PhD Dissertation, Kansas State Univ., 1989.

[18] Shriver, B. and Wegner, P. (editors): *Research Directions in Object-Oriented Programming*, MIT Press, 1987.

[19] Zisman, M.D.: *Representation, Specification and Automation of Office Procedures*, PhD Dissertation, Univ. of Pennsylvania, 1977.