INCREASING FLEXIBILITY OF SIMULATION BY REFORMULATIONS

JERZY A. BARCHANSKI

Department of Computer Science
Brock University
St.Catharines, Ontario, L2S 3A1
CANADA

ABSTRACT
This paper presents an approach to object-oriented simulation of communications protocols based on some concepts borrowed from the field of artificial intelligence. A protocol entity is represented by sets of objects, functions and relations. Reformulation of the protocol entity enables flexible simulation of complex communications protocols at different levels of detail. A proposed simulation language is based on a first-order predicate logic with temporal arguments.
KEYWORDS: abstraction, communication protocols, object - orientation, simulation language.

1. INTRODUCTION

Object-oriented simulation is based on the concepts of object, class, inheritance and message passing. An object is an entity that combines the properties of data and procedure. The interaction between objects is done via message passing. An object responds to a message by executing one of its methods. Objects are organized into classes, i.e. in sets of objects sharing the same properties and the same behaviour. The objects belonging to a class are called class instances. Classes can be organized in inheritance lattice.

Compared with more traditional approaches, object-oriented simulation languages have several advantages, mainly due to a clean data and procedure organization. They support data abstraction, increase the modularity of programs, and due to property inheritance, avoid redundant declarations or specifications.

Conventional object-oriented simulation, as defined above, has a number of limitations, too.

If offers a relatively low level of abstraction, one type of relationship (is-a), one type of inference (inheritance), and one type of representation structure.

Low level of abstraction means that there is an important semantic gap between the real world and its representation. The real world is complex in essence - its representation requires usage of many subtle and rich concepts mixed together or interlinked by various types of relationships (e.g. is-a, is- part-of, has-parts, is-connected-to, etc.). Each type of relation requires support of a specific type of inference mechanism (e.g. inheritance, compositionality, transitivity, etc.). This contrasts with the simplicity of the features provided by conventional object-oriented systems: object, class lattice, inheritance...

The aim of the research described in the following is to fill this semantic gap by proposing some extensions to the conventional object-oriented representations which enable forward and backward simulation at different levels of detail. The proposed approach is based on some concepts borrowed from the field of artificial intelligence (AI). As an example we will use object-oriented simulation of a communication protocol.

In AI knowledge about a world is represented usually in terms of objects presumed or hypothesized to exist in the world and their interrelationships [5]. The notion of object is quite broad. Objects can be concrete (e.g.computer) or abstract (e.g. set of numbers). Objects can be primitive or composite. Not all objects of a world need be considered - the set of objects being considered is often called a universe of discourse. There are different ways to conceptualize a world and to formalize knowledge about it. Some conceptualizations are more appropriate for knowledge formalization than others. While there is no comprehensive criterion for appropriateness of a conceptualization, there are some issues which should be considered. One such issue is the grain size of the objects associated with a conceptualization. Choosing too small a grain can make knowledge formalization prohibitively tedious. Choosing too large a grain can make it impossible. A solution to this is to represent the world at a collection of levels of detail that are related to each other and to simulate it at the level at which we can get the best performance [10]. Better performance can be obtained by using more abstract simulation model, where a single simulation step corresponds to a large number of steps at the lower abstraction levels.

In order to represent a world at a collection of abstraction levels it is necessary to use a declarative language independent of any particular level details. A description in such a language can be used for different tasks - for example, reasoning forward to determine the outputs for some inputs (as in forward simulation), reasoning backwards to deduce the inputs that produced a given output (as in backward simulation). If the original representation is a flat, one-level representation, it is important to reformulate it in order to improve the performance of a simulation. Sometimes it is possible to use descriptions created in the design refinement process. If they are not available, other kinds of reformulation, described below can be used.

2. OBJECT-ORIENTED REPRESENTATION OF A COMMUNICATIONS PROTOCOL

Our representation of a communication protocol will consist of two protocol entities interacting with each other through a common communication medium (Figure 2.1.).

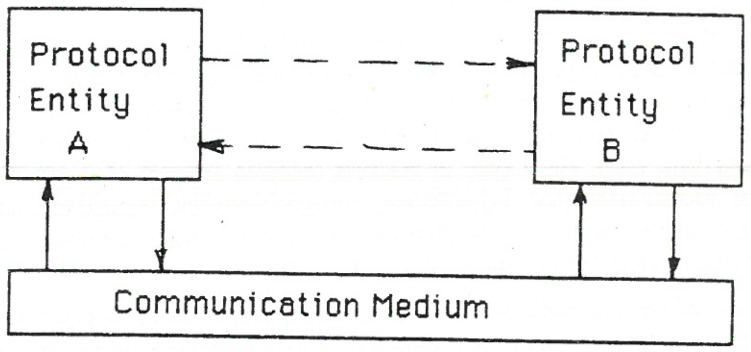

Figure 2.1 Model of a Communication Protocol

A protocol entity may be represented in different ways. For example, we could choose to view it as a single complex function (a black box with a set of inputs and outputs), or we could choose to view it as a composition of functions (a collection of interconnected black boxes), or as a combination of the two. The representation we choose may, in general, be incomplete, ignoring detail that is irrelevant to the task at hand.

In general, a representation D is a three-tuple $< O, F, R >$, where O is a set of objects. F is a set of functions and R is a set of relations. Every object in the representation is associated with a corresponding set of functions. The relationships between various objects are defined by the elements of R.

The set of objects includes the modules, ports and connections. Modules define the components of the entity. Each module has a set of input and/or output ports, which are the only points through which it can communicate with its environment. Communication between modules is defined by connections which relate the values of the ports at its endpoints. State variables are used to define a partial history of the values at ports, or the internal state of modules. Modules, ports and connections can be composite. The subparts of a module are its submodules and their connections. The subparts of a port are its subports, and the subpart of a connection is its submodule. These objects can be decomposed recursively down to a primitive set of modules, ports and connections and grouped in different class taxonomies.

Elements of F define the behaviour of the modules, ports and connections. The behaviour of a module defines the temporal relations between the values of its ports and state variables. For a module with multiple outputs/state variables, a separate function is defined for each output/state variable. For a composite module we have a set of functions (one for each output and state variable) defining its behaviour at the high level, and a composition of functions defining it at the next lower level (and so recursively down to a primitive set of functions). Similarly, the behaviour of a connection specifies the temporal relationship between the values at the ports of its endpoints. The behaviour of a port specifies the temporal relations between its value and the values of its subports.

Members of R define the type of an object in O, and the relationships between these objects, e.g. the relations: module, port, connection, submodule, subport, subconnection, and connected. The first four relations define the type of an object to be a module, port, or connection (is-a relations). The next three relations are of the type is-part-of. A submodule relation defines a part of a module, a subport relation defines a part of a port and a subconnection relation defines a part of a connection. Finally, the "connected" relation associates objects at the endpoints of a connection.

3. REFORMULATIONS OF PROTOCOL ENTITY REPRESENTATION

Reformulation involves translating one representation $D = <O, F, R>$ into another representation $D' = < O', F', R' >$. The reformulated representation D' must be a correct representation of the same protocol entity. By reformulating representation we are changing the way we wish to view the protocol entity. That is, we can choose to view the entity as being composed of a different set of objects, and we can choose to view new relations between these objects.

In reformulating a representation $D = <O, F, R>$ into another representation $D' = <O', F', R' >$ we can either abstract/refine the existing representation, choose a different partitioning for the representation, or make explicit/implicit the functions and relations between the objects. A brief description of each of these reformulations operations is given below.

Abstraction corresponds to creating new objects in the representation, each of which corresponds to a collection of existing objects in the original representation. In addition, we can define new functions (e.g. behaviour) and relations (e.g. connections between objects) for the newly created objects. In abstracting a representation we do not throw away any information - the objects that are subparts of the newly created objects are still a part of the representation.

Refining a representation is the inverse of abstraction, and involves creating a collection of objects that refine an existing object in the original representation. For example, we can refine a representation by defining the substructure of an existing primitive object.

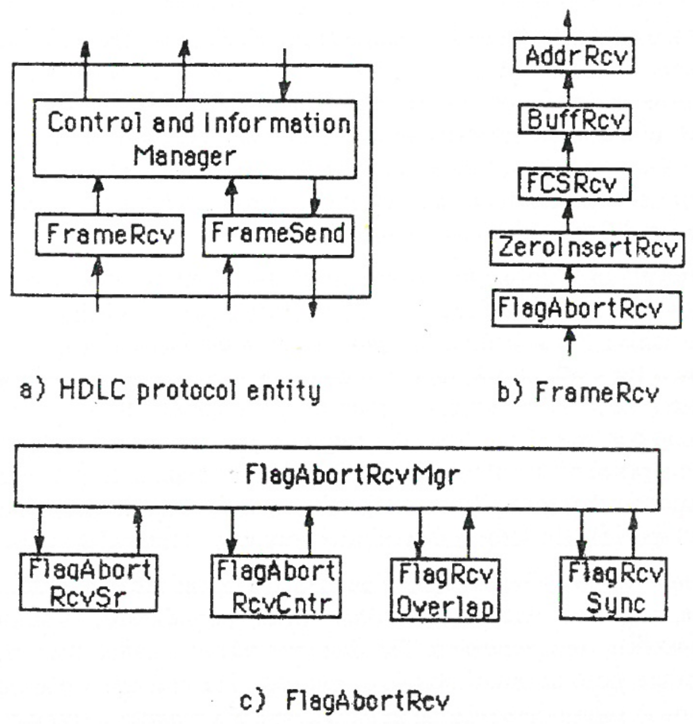

Figure 3. Step-wise Refinement of the HDLC Protocol Entity

Repartitioning a representation involves choosing a different set of objects for a representation such that the primitive objects in the new and old representation are the same.

Finally, reformulating a representation to make knowledge explicit/implicit selects a different space/time tradeoff. All facts that could be deduced in the old representation can still be deduced in the new representation, and vice versa. However, some facts can be deduced more or less efficiently.

As an example of reformulation, let us show a step-wise refinement of the HDLC protocol entity representation, fully described in [2]. At the top level the HDLC Protocol Entity can be represented as a black box with some inputs and outputs connecting the Entity with the lower and upper layer entities. We can reveal the structure of the top level representation by refining it into the interconnection of three major modules - FrameRcv, FrameSend and CIMgr (Figure 3.a.). Beside the same input/output signal streams as in the top level representation, there are three internal signal streams. For each of the modules a description of its behaviour is provided. In the next step of refinement, the internal structure of each of the modules is revealed. For example, the internal structure of the FrameRcv is represented in Figure 3.b. It consists of serial interconnection of five submodules - FlagAbortRcv, ZeroInsertRcv, FCSRcv, BuffRcv and AddrRcv.

In the final step of refinement the structure of each submodule is revealed. For example, the FlagAbortRcv consists of five components as shown in Figure 3.c. - FlagAbortRcvSR, FlagAbortRcvCntr, FlagRcvOverlap, FlagRcvSync and FlagAbortRcvMgr.

It is important to realize that after each refinement step we get not only representation of the level structure but a representation of the behaviour of each component as well. Such partial representation can be used for reasoning at this level.

We describe below a general approach to abstraction. Due to the duality of abstraction and refinement, the discussion on abstraction is equally applicable to refinement.

4. ABSTRACTING A REPRESENTATION

Formally, abstraction involves transforming a representation $D = < O, F, R, >$ into a representation $D' = < O', F', R' >$ such that:

$$O \leq O' \wedge F \leq F' \wedge R \leq R'$$

Each newly created module $o' \in O' \setminus O$ (*) corresponds to a collection of interconnected modules

$$o_{i,1},\ldots o_{i,k} \quad O$$

This abstraction defines a new partitioning on top of the existing partitioning of the original representation D. The new functions in $F' \setminus F$ correspond to the functional relationships between the ports of the newly created modules, and between these ports and the ports of the substructure of the new modules. Similarly, the new relations in $R' \setminus R$ correspond to the relations between the newly created modules, and between these modules and their substructure.

We may distinguish five types of abstraction operations: structural, spatial, temporal, value and functional.

Structural abstraction corresponds to the case where a number of lower level objects is mapped into a higher level object.

Spatial abstraction corresponds to the case where a value at the abstract level corresponds to a collection of values at the lower level, each for a different port at the lower level.

Temporal abstraction corresponds to the case where a single value at the abstract level corresponds to a collection of values at the lower level, each at a different point in time.

Value abstraction corresponds to the case where the set of values at the lower level

*) $O' \setminus O$ means difference between sets O' and O

is partitioned into equivalence classes such that all values in the same equivalence class at the lower level map to a unique value at the abstract level.

Finally, functional abstraction corresponds to the case where we reify a new function (give it a name and define its properties), and define the behaviour of the new objects in terms of this function.

We will consider in the following structural and temporal abstractions in detail, as they seem to be of most utility to our purposes.

4.1. Structural Abstraction

In defining a structural abstraction the vocabulary for describing the behaviour of a newly created module is the same as the vocabulary for describing the behaviour of its subparts, i.e. there is a one-to-one mapping between a value at the abstract level and a value at the lower level. An example of structural abstraction for the HDLC protocol entity is given in Figure 4.1.

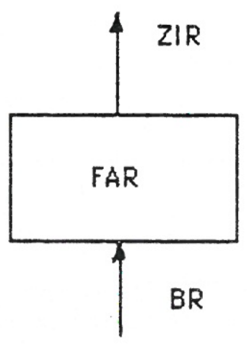

Figure 4.1. Abstraction of the FlagAbortRcv Module

The original representation was defined in terms of a collection of 5 submodules, while the abstracted representation aggregated this structure to define the module FlagAbortRcv and its properties. The set of objects in the original representation O is of five submodules, their ports and sixteen connections.

The functions in the original representation F define the behaviour of each submodule and connection. Similarly the relations in the original representation R define the type of each component, and define the endpoints of the connections.

The set of objects in the structurally abstracted representation includes the existing objects in the original representation, and the new objects corresponding to the FlagAbortRcv and its ports. The new representation has one additional function for the behaviour of the FlagAbortRcv, and 3 additional relations that define the type of the FlagAbortRcv and its ports. A formal description of the new representation is given below:

$$O' = O \cup (FAR, BR, ZIR)$$
$$F' = F \cup (behav_{FAR})$$
$$R' = R \cup (module(FAR), port(BR, FAR), port(ZIR, FAR))$$

The function $behav_{FAR}$ denotes the behaviour of the FlagAbortRcv. The relation module(FAR) defines the object FAR to be a module, and similarly the relation port(BR, FAR) defines the object BR to be a port of the module FAR.

The behaviour of submodules, connections, and the FAR module define the input/output relationships that exist between values at their ports.

4.2. Temporal Abstraction

Temporal abstraction corresponds to the cases where a single value at the abstract level at a point in time is related to a collection of values at the lower level, each at a different point in time (Figure 4.2). It lets us view operations of a protocol entity with different time granularities. The set of objects and relations in the temporally abstracted representation is the same as in the structurally abstracted representation. The temporally abstracted representation has one additional function for the temporally abstracted behaviour of the module as a whole.

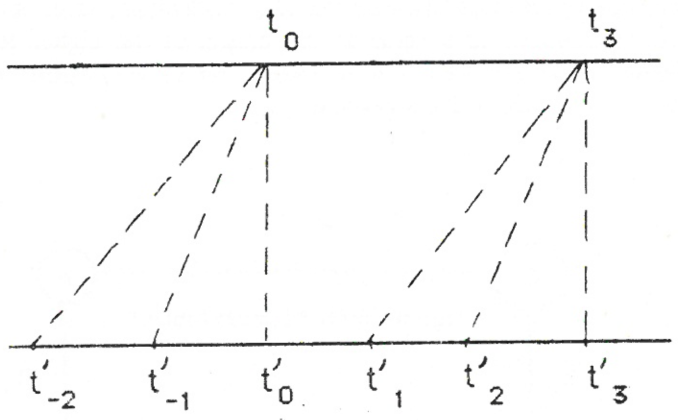

Figure 4.2. Temporal Abstraction

5. CORRECTNESS OF REPRESENTATION REFORMULATION

Reformulation of a representation lets us decrease the size of the representation and improve performance of a simulation. Having the different formulations related to each other permits shifting from one abstraction level to another, whenever it is computationally advantageous to do so. However the computational savings in using the more appropriate formulation cannot come at the expense of correctness. The behaviour of each component of the representation must be correct, and the behaviour of a composite component must be a correct abstraction of the composition of the behaviours of its subparts. However, the two behaviours need not be the same, since in defining the behaviour of more abstract components we will usually want to ignore the details that are irrelevant at that level. There is a tension between the two competing goals of correctness and ignoring details. Maintaining correctness must strike a balance between these conflicting goals.

In abstracting the behaviour of a collection of components at a given abstraction level it is often useful to simplify the descriptions by approximating the exact behaviour at the lower level. For a hierarchical representation these approximations combine with each other for each level in the hierarchy. We must define the properties of the lowest level behaviour that we wish to preserve in order to decide if a given design is correct.

The structural formulation at a given abstraction level is correct if every module, port, connection and state-variable of the representation has a correspondent at the adjacent lower abstraction level. However, the structural correspondence need not be complete, since we may wish to ignore some details.

In addition to the structural correspondence, the behaviour of a representation at a given abstraction level must be a correct mapping of the behaviour of the representation at the adjacent lower level. In order that the representation should be correct we must define a homomorphism between the behaviour of the representations at the adjacent levels.

This homomorphism is defined using a commutative diagram, as shown in Figure 5.1. [14]. In this diagram, all paths with the same starting and ending nodes must produce identical results. The homomorphism is defined by the function h_d which maps a value at the input of a lower level component to a value at the input of the higher level component representation, and the function h_r, which maps a value at the output of the lower level component to a value at the output of the higher level component representation. There are two important parts of the value of a port - the actual value, and the time at which this value is present.

$$\begin{tikzcd}
d \arrow[r, "f", "\text{higher level representation}"'] & r \\
d' \arrow[u, "h_d"] \arrow[r, "f'", "\text{lower level representation}"'] & r' \arrow[u, "h_r"']
\end{tikzcd}$$

Figure 5.1. Homomorphism Between Adjacent Abstraction Levels

Let the lower level function be f', and the higher level function be f. The function f' maps arguments in its domain d' to its range r'. Similarly, the function f maps arguments in its domain d to its range r.

In general, the homomorphism function h_d maps a collection of values $v1_{t1},...,vi_{ti}$ from d' (the subscripts denote the time), to a value v_t in d, and similarly for the function h_r. In order that the representation should be correct abstraction, the following commutative relation must hold true:

$$f(h_d(v1_{t1}),.....,h_d(vi_{ti})) = h_r(f'(v1_{t1},.....,vi_{ti}))$$

In addition, for hierarchical representation, the transitive closure of the homomorphism functions at each abstraction level must satisfy the commutative relation as well.

We will consider in the following reformulation involving structural, functional and temporal abstraction. To accurately model behaviour of a structurally abstracted representation, such as in the example of the FlagAbortRcv in p.5.1., we must enumerate the input and the output combinations since the temporal relation between them is a function of the specific values at the input ports. Although such description of behaviour is accurate, it precludes a symbolic definition (functional abstraction) of behaviour. However, it is possible to provide a symbolic description if we are willing to approximate the exact temporal behaviour at the lower level, by replacing a sequence of output values corresponding to different time instances by a value of the function computed for the last time instance. Consequently, for certain inputs, the abstracted behaviour will specify some outputs with excessive delay.

In general, in approximating the temporal behaviour of a collection of modules at level i, there is a minimum/maximum delay from the inputs of the collection of modules to the outputs of the collection of modules $\delta_{min,i} / \delta_{max,i}$. An approximation of the temporal behaviour for the collection of modules must specify a delay in this range. The magnitude of the temporal error $\delta_{error,i}$ for any input is bounded by $\delta_{max,i} - \delta_{min,i}$, for abstraction level i. It may be represented in the commutative diagram by mapping a value v at time t at the lower level to the same value v at time abst(t) at the abstract level. The function abst(t) maps all times at the lower level to the same time at the abstract level, except that the times at the lower level $t_{trans} < t < t_{trans} + \delta_{error,i}$ maps to the time point $t_{trans} + \delta_{error,i}$ at the abstract level. The expression t_{trans} stands for the time at which the value changes at the lower level.

In order that this temporal mapping be unique these time ranges must not overlap, i.e. the maximum temporal error must be less than the minimum time between transitions ($t_{min,i}$) at the lower level. Therefore, the following constraint must hold true at all abstraction levels for a hierarchical representation:

$$\delta_{error,i} < t_{min,i}$$

Sometimes it is useful to add an additional constraint which requires that when a value at an abstract level changes to v at time t, then the values at all the lower levels at the same time t must map to this value v (under the transitive closure of the mapping between adjacent levels in the hierarchy). Results of simulation at the higher level depend on the time at which the output of a module changes. When using a more abstract description which is an approximation of the exact behaviour it is better to restrict the inaccuracies to occur at times other than those at which there are transitions at the higher level.

A consequence of this constraint is that in approximating the delay for a collection of modules we must choose the maximum delay $\delta_{max,i}$, since we must ensure that the output of the high- level behaviour does not change before the lower levels. In addition, for the values to agree at the high level transitions we must ensure that the temporal errors at the lower abstraction levels do not add up to greater than the minimum time between transitions of abstraction level i. That is, the following constraint must be true:

$$\sum_{level=1}^{i} (\delta_{max,level} - \delta_{min,level}) < t_{min,i}$$

6. SIMULATION LANGUAGE

6.1. Requirements

For representation of a communication protocol the semantics of a simulation language must map a well-formed sentence in the language to either an object, a function or a relation between the objects of the representation. The language must be general enough to represent arbitrary objects, and arbitrary functions and relations between these objects. As we want to use the same protocol entity representation for forward and backward simulation, the language must allow to do so. For example, a procedural language does not fulfill these requirements - it is adequate for forward simulation, but not for backward simulation. In forward simulation we are interested in propagating information from the inputs of modules to their outputs, which corresponds to computing the results of a function for given arguments. In backward simulation however, we are interested in finding the inputs of a module that could produce a given output. This requires inverting functions, which is extremely difficult if functions are represented as procedures in traditional simulation languages*).

6.2. Logic As A Simulation Language

The above requirements can be satisfied by a language based on a logic. Many advantages come with the use of logic as the representation formalism. First of all, unlike most other representation formalisms, logics come with a formal semantics, giving a precise description of the meaning of expressions in the formalism. Secondly, again unlike many other knowledge representation formalisms, logics have well-understood properties as regards their completeness, soundness, and decidability. For any reasonable logic it is possible to prove that the proof theory is sound. Furthermore, it is possible to establish via formal methods whether a particular logic is complete or not. It is known that, for any reasonably powerful logic, provability is at best a semi-decidable property. Although these results are in themselves sometimes negative (e.g. incompleteness, semi-decidability), the important point is that these properties are known at all. For many other representation formalisms no such results have been obtained.

The next advantage in favour of logic is its expressive power. Two aspects of this must be mentioned. Firstly, the language of logic is not restricted to that of standard two-valued, truth- functional, first-order predicate calculus. Many other logics have been proposed, offering a wide range of expressional and inferential power. A few examples of these are intuitionistic logics, many-valued logics, modal logics, epistemic

---

*) At the very least functions would have to be first-class objects, as they are in Lisp and assembly languages. However, inverting functions is made difficult by side-effects and non- local variable references in the body of a function.

logics and tense logics. The second important aspect of the expressive power of logic is its ability to express what might be called incomplete knowledge, or information about incompletely known situations. We can say that the expressive power of logic determines not so much what can be said, but what can be left unsaid. As a result, one is not forced to represent details that are not known (yet) - a possibility extremely useful for representation reformulation.

As a representation language logic is task-independent, since the information is represented declaratively. That is, there is a precise definition of the interaction between the parts of a description, and in addition, these interactions are only based on pattern matching parts of expressions via unification. The same description can thus be used for different purposes, e.g. forward and backward simulation.

Simulation of communications protocols requires explicit representation of time - so we need a temporal logic, able to represent changes of behaviour dependent on time. There are many different temporal logics - so we have to decide which one to use. We have considered three kinds of temporal logics - modal logic [3, 8], reified logic [7, 11] and first-order logic with temporal arguments (MTA) [6] from the viewpoint of our requirements and have decided to take the MTA logic as a basis of our simulation language [1].

The MTA logic is an approach to representing temporal information in which every ordinary predicate of a logic is supplemented by a special argument (or arguments) for time. The extra argument refers to the time at which the atomic proposition formed with it holds. Temporal predicates for expression of temporal properties and relations, such as duration and ordering, can come in many types and combinations in MTA logic. The basic temporal entities referred to by temporal arguments may be either points or intervals. In order to be able to do temporal reasoning in MTA, one needs to introduce an explicit ordering relation $<$. $t_1 < t_2$ is to be read as t_1 is earlier than t_2. In order to be able to express that something was the case at some unspecified time in the past, one also has to add to the language a special time- constant t_0 referring to the present time. The method of temporal arguments is preferable to the other approaches on computational grounds. One can use the standard efficient techniques for implementing first-order logic. MTA has also an important expressive advantage over modal temporal logic. Although a version of modal logic may use time constants, it is not possible to quantify over them. Quantification over points in time can only be achieved by using the modal operators. In MTA logic, time-points appear as arguments to the predicates, and therefore can be quantified over.

7. PROTOCOL ENTITY REPRESENTATION WITH THE MTA LOGIC

The MTA logic can be used with different approaches to represent behaviour of communication protocols modules - finite state automata, Petri nets, labelled transition systems and others. In the following example the MTA logic is used with a labelled transition system in which transitions between states are described by the general formula:

$$\text{Exists}((t_1,t_2), S_1)\ \&\ \text{Occurs}((t_2,t_3), A_1) \Rightarrow \text{Occurs}\ ((t_4,t_5),A_2)\ \&\ \text{Exists}\ ((t_5,t_6), S_2)$$

In this formula, the connective symbol "$\Rightarrow$" represents the logical implication corresponding to a transition. The first Exists predicate represents a state S_1 holding since the end of the previous transition during time interval (t_1,t_2). The first Occurs predicate represents an action stimulating the present transition and lasting during time interval (t_2,t_3). The Occurs predicate following it with a possible delay represents a resulting action lasting during time interval (t_4,t_5). The second Exists predicate represents a new state S_2, which will hold from the end of the present transition until the beginning of the next transition, during time interval (t_5,t_6) (Figure 7.1.).

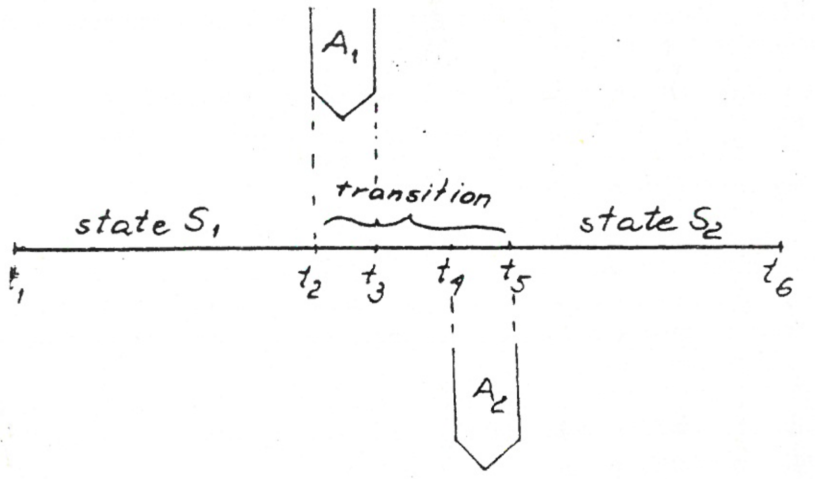

Figure 7.1. Temporal Representation of a State Transition

We have assumed here the most general case in which we deal with different (but non-zero) durations of the action intervals. The resulting action follows the stimulating action with a delay, so we have a non-zero transition time and the state intervals are separated by the transition intervals. Time is counted from the beginning of the previous state until the end of the next state. Time points are temporal variables and are totally ordered (though not shown explicitly in the formula).

We can abstract behaviour represented by this formula, by assuming durationless actions occurring during the transition interval, which may be durationless as well (Figure 7.2.). It still leaves us with the possibility to represent duration of states as time intervals between consecutive stimulating actions. In the final step of temporal abstraction we drop the concept of time altogether - what is left is the total ordering of states only.

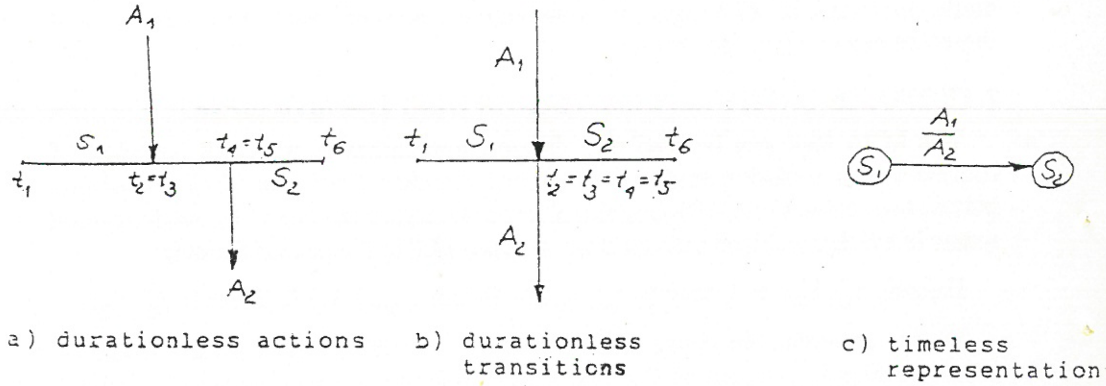

Figure 7.2. Abstractions of Temporal Representation of State Transition

We can consider three different kinds of actions: two stimulating actions - message reception (represented by the Rcv predicate) and an internal action (represented by the IA predicate) and one resulting action - message transmission (represented by the Xmt predicate). We can thus simplify the above notation by introducing specialized predicates for each kind of action:

$$\text{Exists}\ ((t_1,t_2), S_1)\ \&\ (\text{Rcv}((t_2,t_3), M_1)\ \vee\ \text{IA}((t_2,t_3), A_1))\ \Rightarrow$$
$$\text{Xmt}\ ((t_4,t_5), M_2)\ \&\ \text{Exists}\ ((t_5,t_6), S_2)$$

As an example we will describe a behaviour of a protocol module A initiating connection establishment with a protocol module B according to the three way handshake protocol [12]. The state space of this protocol module consists of the following states (Figure 7.3.): CLOSED(x), SYNSENT(x) and ESTAB(x,y), where x and y are the initial sequence numbers selected by the communicating modules.

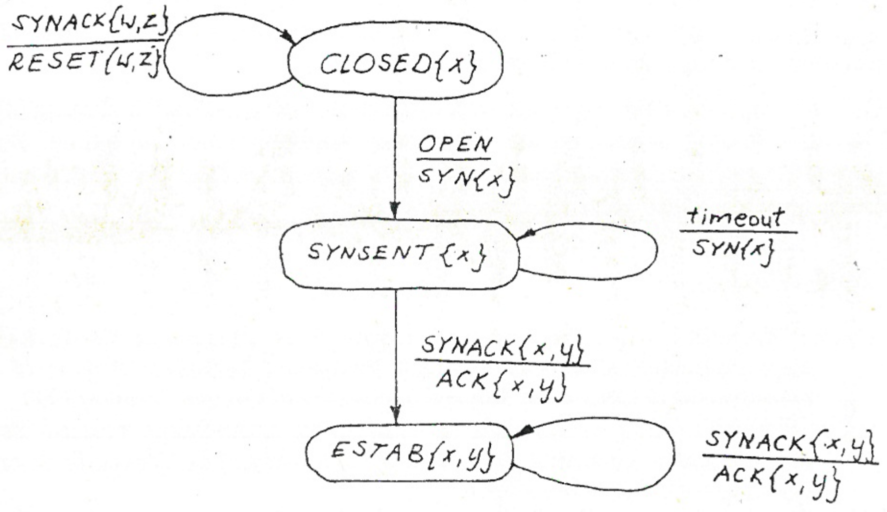

Figure 7.3. State Transition Diagram of a Connection Establishment Protocol

In state CLOSED(x), upon receiving a user request OPEN. A transmits a connection initiation message SYN(x) to B and enters state SYNSENT(x):

$$\text{Exists}\ ((t_1,t_2), \text{CLOSED}(x))\ \&\ \text{Rcv}((t_2,t_3), \text{OPEN})\ \Rightarrow \text{Xmt}\ ((t_4,t_5), \text{SYN}(x))\ \&$$
$$\text{Exists}((t_5,t_6), \text{SYNSENT}(x))$$

However, if a SYNACK is received before the user request, a RESET will be returned:

$$\text{Exists}((t_1,t_2), \text{CLOSED}(x))\ \&\ \text{Rcv}((t_2,t_3), \text{SYNACK}(w,z))\ \Rightarrow$$
$$\text{Xmt}((t_4,t_5), \text{RESET}(w,z))\ \&\ \text{Exists}((t_5,t_6), \text{CLOSED}(x))$$

In state SYNSENT(x), if a message SYNACK(x,y) is received which represents a response to the connection initiation SYN(x), A transmits an acknowledgment ACK(x,y) to B and enters state ESTAB(x,y):

$$\text{Exists}((t_5,t_6), \text{SYNSENT}(x))\ \&\ \text{Rcv}((t_6,t_7), \text{SYNACK}(x,y))\ \Rightarrow$$

$Xmt((t_8,t_9), ACK(x,y)) \& Exists((t_9,t_{10}), ESTAB(x,y))$

If the message SYN(x) remains unacknowledged beyond the time-out period b, it will be retransmitted:

$Exists((t_5,t_5+b), SYNSENT(x)) \& Rcv((t_5,t_5+b) \neg SYNACK(x,y)) \Rightarrow$

$Xmt((t_5+b,t_6), SYN(x)) \& Exists((t_6,t_7), SYNSENT(x))$

In state ESTAB(x,y), if a duplicated SYNACK(x,y) arrives, a duplicated ACK(x,y) is returned:

$Exists((t_9,t_{10}), ESTAB(x,y)) \& Rcv((t_{10},t_{11}), SYNACK(x,y)) \Rightarrow$

$Xmt((t_{12},t_{13}), ACK(x,y)) \& Exists((t_{14},t_{15}), ESTAB(x,y))$

8. CONCLUSIONS

The approach described above is one of the many proposed applications of artificial intelligence to simulation [9,13]. The specific features of our approach are object-oriented representation and reformulation methodology, a representation language based on first order logic with temporal arguments and usage of the same representation for forward and backward simulation.

We have implemented the MTA in a version of Prolog called CS- Prolog [4]. It is a parallel Prolog using notions of process, communication and time. Being developed for multi-transputer networks, it lets us represent the distributed nature of communication protocols in a natural way.

REFERENCES

1. BARCHANSKI, J.A., **Issues and Choices of Temporal Logic-Based Representation of Communication Protocols,** Technical Report of the Department of Computer Science, University of Ottawa, January 1990.
2. BARCHANSKI, J.A, **Framework for Structured, Logic-Based Protocol Entity Representation** Technical Report of the Department of Computer Science, University of Ottawa, July 1989.
3. CAVALLI, A.R and HORN, F., **Proof of Specification Properties by Using Finite State Machines and Temporal Logic,** Proceedings of the 7th Symposium on Protocol Specification, Testing and Verification, 1987.
4. FUTO, I. and KACSUK, P., **CS-Prolog on Multi-Transputer Systems, Microprocessor and Microsystems,** Vol.13, No.2, March 1989.
5. GENESERETH, M.R. and NILSSON, N.J., **Logical Foundations of Artificial Intelligence,** MORGAN KAUFMANN, 1987.
6. HAUGH, B.A., **Non-Standard Semantics for the Methods of Temporal Arguments,** Proceedings of the 1987 International Joint Conference on Artificial Intelligence, pp.449-455.
7. REICHGELT, H., **Semantics for Reified Temporal Logics,** Department of AI Research Paper No. 299, University of Edinburgh, 1986.
8. **Second International Workshop on Protocol Specification, Testing and Verification,** Session on Temporal Logic, Idyllwild, CA, 1982.
9. SHIRATORI, N. et al, **An Intelligent User-Friendly Support System for Protocol and Communication Software Development,** Proceedings of the 8th International Symposium on Protocol Specification, Testing and Verification, 1988.
10. SINGH, N. **Exploiting Design Morphology to Manage Complexity,** Ph.D. Thesis, Stanford University, 1985.

10. SINGH, N. **Exploiting Design Morphology to Manage Complexity,** Ph.D. Thesis, Stanford University, 1985.
11. SHOHAM,Y. **Reified Temporal Logics: Semantical and Ontological Consideration,** Proceedings of the 1986 European Conference on Artificial Intelligence.
12. SUNSHINE, C. and DALAL, Y., **Connection Management in Transport Protocols,** COMPUTER NETWORKS 2(6), December 1978, pp. 454-473.
13. WIDMAN, L.E. et al, **Artificial Intelligence, Simulation and Modelling,** JOHN WILEY, 1989.
14. ZEIGLER, B.P., **Multifacetted Modelling and Discrete Event Simulation,** ACADEMIC PRESS, 1984.