Annales Univ. Sci. Budapest., Sect. Comp. 17 (1998) 253-268

A SPECIFICATION TECHNIQUE FOR SCHEDULING THE METHODS OF CONCURRENT OBJECTS

L. Kozma and É. Rácz (Budapest, Hungary)

Abstract. Object-oriented programming methodology has been rapidly accepted by the experts. The most critical applications can often involve concurrency, but several problems have been identified with the integration of inheritance and concurrency. A lot of methods can be defined on a concurrent object but some operations make sense only when the object is in certain states. Synchronization mechanisms are necessary to delay such requests and these requests can be processed when the state of the object makes them safely possible. This paper deals with a similar problem called scheduling when some method of a concurrent object is selected for execution to achieve more effective functioning in some sense. We apply the graphical interval logic, GIL, for specifying scheduling the methods of concurrent objects.

1. Introduction

Object-oriented programming methodology has been rapidly accepted by the experts. The need for data abstraction to promote program modularity and the claim for developing correct programs for real applications underlie this rapid acceptance. Recent advances in hardware technologies can assure extremely low hardware failure rates for devices. Unfortunately, technologies for software engineering have not matched this advance. The most critical real applications can often involve concurrency. Reasoning about concurrent systems is considerably more difficult than reasoning about sequential systems. Moreover nondeterminism and the reactive character of these concurrent applications make them hard to develop, to validate and to test.

Supported by the Hungarian Science Grant (OTKA), Grant Nr. T017800.

It turns out that concurrency is a natural consequence of the concept of objects, but several problems have been identified with the integration of inheritance and concurrency. The first problem is that the term inheritance has two primary meanings: a mechanism by which object implementations can be organized to share codes; and a classification of objects based on common behavior or common external interfaces. So the term inheritance has been resource of confusion. The fact is that the inheritance structure used for code sharing and the conceptual subtyping hierarchy originated from specialization are not the same things. They lie on different levels of abstraction in the system: inheritance is concerned with the implementation of the classes, while the subtyping hierarchy is based on the behavior of the instances and it describes how an object can be seen from the outside, by other objects [1,4,5,15, etc.].

The additional problems are related with difficulties in integrating sharing protocols in systems based on actor model and with the fact that synchronization constraints often interfere with inheritance. Concurrent objects can be defined as entities that encapsulate data and operations into a single computational unit. Models of concurrent computation based on objects must specify how the objects interact and different design concerns have led to different models of communication between objects.

A common semantic approach to modeling objects is to view the behavior of objects as functions of incoming communications. This is the approach taken in the actor model [2]. A lot of *methods* can be defined on a concurrent object, but some methods make sense only when the object is in certain states. Synchronization mechanism is necessary to delay such requests and these requests can be processed when the state of the object makes them safely possible. Several language constructs and specification methods were suggested for solving synchronization problems of concurrent object-oriented programming [3, 6 - 12, 14, etc.].

This paper deals with a similar problem called scheduling the methods of concurrent objects. The notion scheduling is used here, when some method of a concurrent object is selected for execution to achieve more effective functioning in some sense. We describe an application of the specification technique suggested in [13, 17]. A graphical interval logic (GIL) is used to specify the scheduling of the methods of concurrent objects. GIL is a visual temporal logic in which formulas resemble the informal timing diagrams familiar to the designers of hardware systems and to software engineers as well. A well-known scheduling example, the "disk scheduler" is specified to illustrate our idea.

The paper provides an overview of the actor model and its extensions in Section 2. A technique is suggested for specifying scheduling policies in Section 3. We give an example in Section 4. Some concluding remarks are discussed in Section 5.

2. Models of concurrent objects

There are different kinds of models of objects: the actor model introduced by G. Agha in [2]; the process model of objects applied in programming languages ABCL, POOL, Concurrent Smalltalk, etc.; other models used in concurrent logic languages and in functional programming languages. The traditional sequential process model allows arbitrary control structures to be specified within the body of a given object, but on the other hand this model has several disadvantages. For instance the sequential process that is optimal on one architecture may not be so on an other one with different characteristics [1]. From point of view of semantics the behavior of all concurrent object-oriented languages can be modeled by actors whether they use a process model or an other one.

The basic *actor model* is object-based and not object-oriented. This means that the model considers each object to include a behavior and a mailbox, but neither the behavior nor the mailbox can be objects. This makes the inheritance hard to handle in the basic actor model. The behavior of an object consists of a so called script and a set of acquaintances. The script is a code body defining the response of the object when a message is received. When the object processes a communication (an answer for a message), it determines the behavior that will be used to process the next communication. The object may also send messages to specific target objects and create new objects. The target object in this case can be any object referenced in the received message, any acquaintance of the objects, and any object created during execution of the script. An object's mailbox holds messages that have arrived but not yet been processed. One of the most important features of object-oriented languages is the inheritance. In basic actor model the script is a monolithic code and this leads naturally to delegation protocols, but delegation is not the most effective approach for code sharing [1,3, 4, etc.]. The actor model has to be extended to directly support inheritance and make it object-oriented.

The *reflective model* of objects fulfills the above conditions. This model suggested in [3] extends the basic actor model in the following way. An object is composed of four other objects: the meta, the container, the processor and the mailbox object. The composition can be recursive. The meta object manages the three other resources of an object. The container represents the acquaintances and the script is a set of slots. The script is decomposed into methods, each of them can be an object similar to the other acquaintances of the object. The processor object is responsible for locating an appropriate method to handle a communication, performing the execution, and coordinating behavior (state) change. The mailbox object holds messages that have not yet been processed as in the basic model, but in the reflective model it may be used to mediate different scheduling policies that determine the next message to be processed. To support sharing protocols between objects the reflective model separates the message passing activities into two levels. The object level deals with messages between objects, while the reflective level deals with messages between containers. The containers are responsible for implementing the sharing protocol as well.

Using reflective model of objects we can separate and localize not only the synchronization schemes, but also the scheduling policy from the main bodies of methods. This allows dynamic operations on the methods themselves in order to control the messages acceptable by an object. As notions scheduling and synchronization are very close to each other, the scheduling policies can be inherited in a similar way to the synchronization schemes suggested by [7]. The advantages of this model are as follows:

(1) The same language features can be used for implementing a scheduling policy as for implementing synchronization code.

(2) The manner we re-use the synchronization code is the manner we re-use the scheduling code.

(3) The reflective model offers a high degree of encapsulation and re-use for synchronization code.

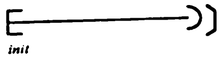
A useful technique is shown during the next sections for specifying scheduling policies.

3. A technique for specifying scheduling policies

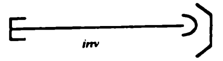
Experiences have shown that temporal logic specifications are too complex to be readily understood. This complexity comes from the need to establish the temporal context where the properties, such as invariance and bounded liveness, must be held. Interval logics can decrease this complexity by defining temporal intervals to represent such contexts. The GIL was developed as a language and a tool set for the behavioral specifications and verifications of concurrent software systems [13,17]. We apply formulas of GIL for specifying scheduling methods of concurrent objects to achieve more effective functioning in some sense.

GIL allows the user to construct arbitrary intervals of time and to express properties that apply to those intervals. GIL also uses predicates that may have different truth values at different points of time to represent properties of systems that change with time. Like in linear-time temporal logics, in

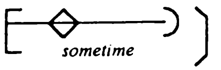
GIL the computations of a system are modeled as linear sequences of states, where every state has assigned truth values to the predicates included in the specification. The basic construct of GIL is the interval representing a finite or infinite computation of a system. An interval can be represented by a left-closed right open line segment: [). The time progresses from left to right inside an interval. Every interval has an initial state, but since contexts are infinite an interval has no a final point. The individual states are represented by points on a line segment with the horizontal dimension showing progression through the context. The basic interval formula



asserts that a formula *init* holds at the first state of the interval. The derived operator *henceforth* can be represented by the interval formula

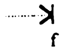


It asserts that the formula *inv* holds at every point in a context. The eventually property *sometime* can be formulated as follows:

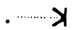


having the meaning that the property *sometime* holds at some state within the interval.

The intervals in the above formulas could represent the entire computation of the system, or they could be extracted from larger intervals using search operators that locate, starting from a specified state, the earliest state at which a property called target formula holds. The following search primitives are defined in GIL:



represents a search to a target formula f;



represents the search to the right end of the context. The dot represents the starting point of the search. The search fails if the target does not hold at any future point in the context. The future always includes the present. A search to the right end of the context permits the specification of a tail interval;

represents a strong search operator. The strong search operator is required not to fail and it is denoted by double arrowhead. A strong interval formula is an interval formula that is required not to be empty and is drawn by a bold line:

The star * denotes a so-called point operator. This operator appears directly below the point located by the final search in a sequence of searches and constructs the tail interval that starts with the point so located. The target formula of the searches and the formulas that are asserted to hold on intervals can be arbitrary GIL formulas, so GIL formulas can be constructed in a hierarchical or nested way. GIL formulas can be read from left to right and from top to bottom [13, 17, etc.].

4. Disk scheduler, an example

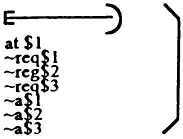
A well-known scheduling example is the disk scheduler problem [10]. In this case a concurrent object is defined for controlling a disk device. For the seek of simplicity, it is assumed that a single operation access is defined for the object. The operation access has only one parameter, the number of the track to be accessed. What is desired is that the average waiting time for access be minimized. This is accomplished by having the head of the disk sweep continuously in one direction, accessing each track it encounters for which an access message has been arrived, until no more requested tracks remain in the direction being swept. Then the head reverses its direction and sweeps back, again accessing requested tracks as it encounters them. The essential idea is that at any given moment the next track to be accessed is the closest one to the actually accessed track in the direction currently being swept.

4.1. Specification of the disk head scheduling

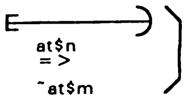
For simplicity, we consider a disk with three tracks. The following state predicates are used:

- The predicate *at$n* is true when the head of the disk is at track n and false when it is not, for n=1,2,3. We suppose that the number of track is growing up from left to right.
- The predicate *movingforward* (denoted by *mfwd*) models the situation, when the head leaves a track and the direction of moving is forward or backward. If *movingforward* is true then the head is moving to forward and if *movingforward* is false then the head is moving to backward.
- The predicate *access$n* (denoted by *a$n*) is true when the head is at track n, in this case the contents of the track n can be accessed.
- The predicate *req$n* is true when there is an outstanding request for accessing the contents of the track n.
- The predicate *arriveforward* (denoted by *afwd*) is true whenever the head arrives at a track from the forward direction and it remains true at least until the head departs the given track.

Init.



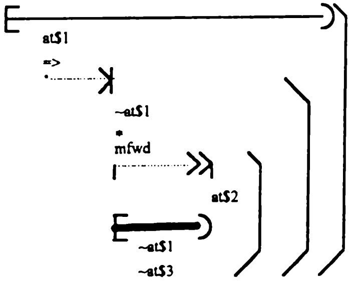
Exclusion$n$m (1 ≤ n ≤ m ≤ 3).



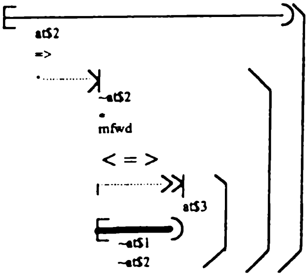
The formula *Init* means that the disk head begins operation at the track 1, there are no requests for services and all tracks are not accessed.

The formula *Exclusion$n$m* means that the head is never at two different tracks simultaneously.

FwdFrom$1.



FwdFrom$2.

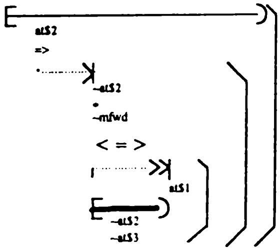


The formula *Fwdfrom$1* means that the head moves forward when it departs the track 1, arriving at track 2 without first sweeping any other tracks.

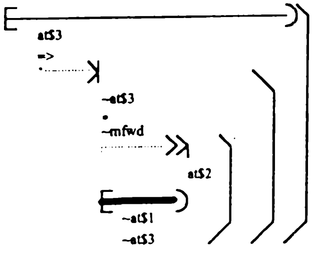
The invariant of this formula expresses the following. The track 1 is the location that the head has just left. The formula asserts that the head is moving forward at every such point and that it reaches the track 2 before either of the other tracks. The strong search requires that the head eventually arrives at the track 2 and the strong interval requires that it takes some time to arrive at track 2.

The formula *FwdFrom*\$2 expresses that the head moves forward when it departs the track 2 precisely if it moves directly to the track 3.

BwdFrom$2.



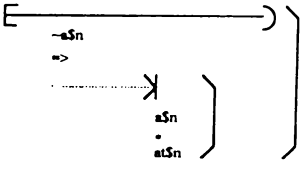
BwdFrom$3.



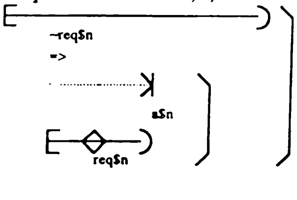
The formula *BwdFrom*\$2 expresses that the head moves backward when it departs the track 2 precisely if it moves directly to the track 1.

The formula *BwdFrom*\$3 means that the head moves backwards when it departs track 3 arriving at track 2 without sweeping any other tracks first.

Safeaccess$n n=1,2,3.



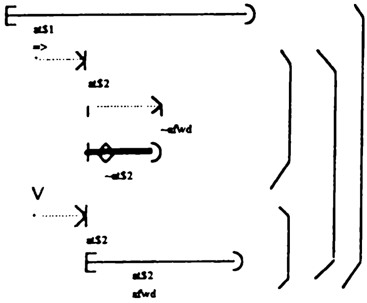
ReqService$n n=1,2,3.



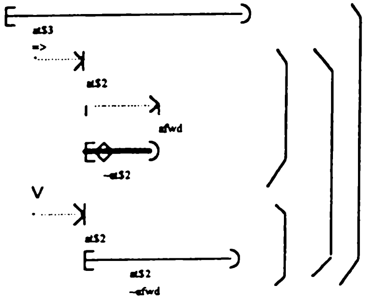
The formula *Safeaccess*\$*n* means that the contents of a track can be accessed only when the head is at that track.

The formula *ReqService*\$*n* expresses that the contents of a track can be accessed only in response to a request for service of the track.

ArriveFwd.



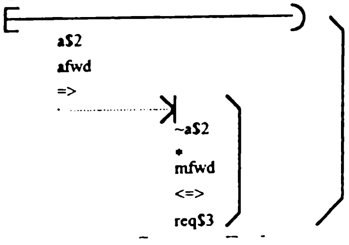
ArriveBwd.



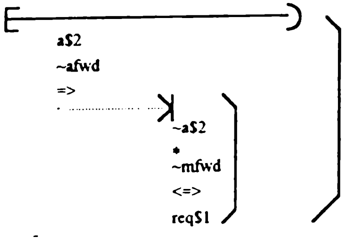
The following formula *ArriveFwd* expresses whenever the head arrives at the track 2 from the track 1 predicate *afwd* is true, and remains true at least until the head departs the track or it remains true forever inside the context if the head does not depart the track 2.

The formula *ArriveBwd* tells whenever the head arrives at track 2 from the track 3 *afwd* is false, and it remains false at least until the head departs the track 2 or the head remains at track 2 forever.

ContinueFwd.



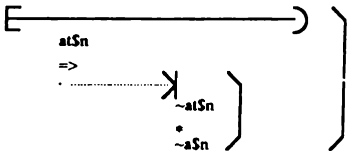
ContinueBwd.



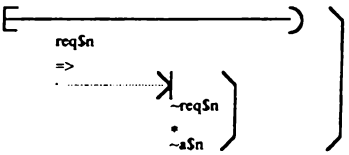
The formula *ContinueFwd* means if the head is moving forward when it arrives at the track 2, it continues moving forward when it departs the track 2 precisely if someone requires service for contents of track 3 by the time the head departs.

According to the formula *ContinueBwd* if the head is moving backward when it arrives at the track 2, it continues moving backward when it departs the track 2 precisely if a service is required for the track 1 by the time the head departs. The above two formulas require that, once the head starts sweeping in a given direction, it changes direction only if no one requires service at a track in that direction.

SafeDepart$n n = 1,2,3.



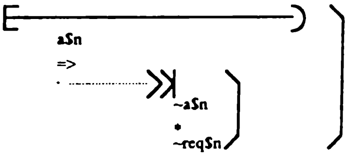
WaitService$n n = 1,2,3.



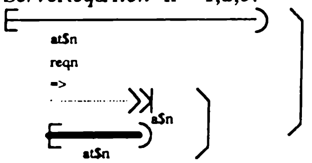
The formula *SafeDepart$n* states that the head departs a track only when the access is finished.

The formula *Waitservice$n* states that a request for a service at a track is only canceled if the track is being serviced (the access is taking place).

CancelService$n n = 1,2,3.



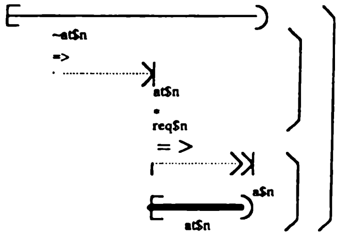
ServeReqsAt$n n = 1,2,3.



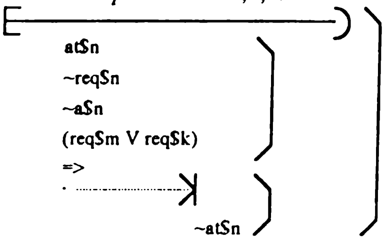
The formula *CancelService$n* expresses that an access always takes place during finite time, and the request for service at the current track is canceled.

The formula *ServeReqsAt$n* states that if a service is requested for a track while the head is at the track, then the access takes place before departing the track.

ServeReqsOnArrival$n n = 1,2,3.



NoServeDepart$n n = 1,2,3.



The above formula *ServeReqsOnArrival$n* states that, if a request for an access of a track is arrived by the time the head reaches the track, then the access takes place before departing the track.

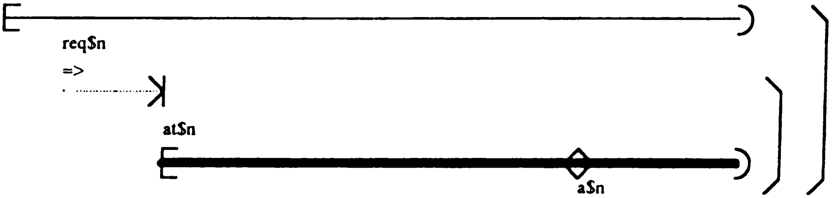
The formula *NoServeDepart$n* ensures that the head departs a track without first accessing its content whenever nobody requires service of the track and a service is required for a different track (where m and k are used to denote the other tracks $\{m, k\} = \{1, 2, 3\} \setminus \{n\}$).

4.2. Analysis of the specification

The use of formal specifications is very important both in the hardware design and in the software technology. The specification can be analyzed for potential consequences and this helps better understanding of the design. Analysis can demonstrate that specification ensures higher-level system requirements correctly. A graphical proof system presented in [13] could be used for checking the validity of the consequences. The major advantage of the proof technique [13] is that a proof can be represented using pictures that show the temporal flow of the argument. Using the time line representation allows one to align appropriate points in the picture. This helps us to see the points where invariants are being instantiated, the relationships between different points and intervals, etc.

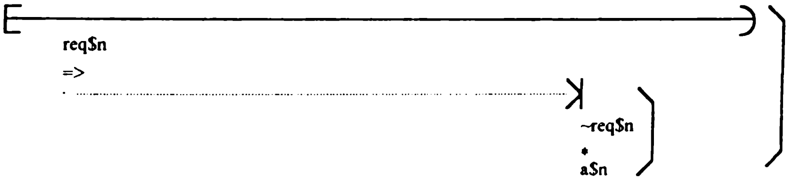
The specification of a disk head scheduling is very similar to the specification of the elevator [13]. The main difference is the following. The specification of the disk head scheduling ensures: if a request arrives for a track while the head is at that track then the request is always serviced before departing the track, so this specification is not starvation-free. The following theorem expresses this fact and its proof shows that a complex proof can be splitted into more manageable steps [13].

Theorem. *Service$n*.

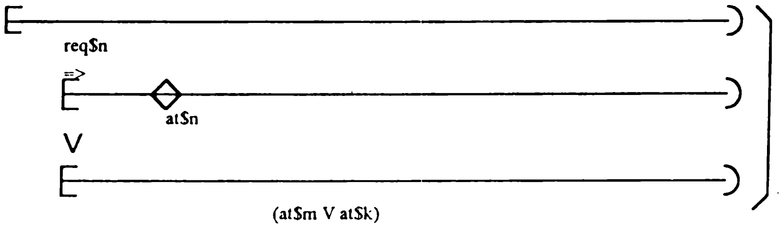


Proof.

WaitService$n:



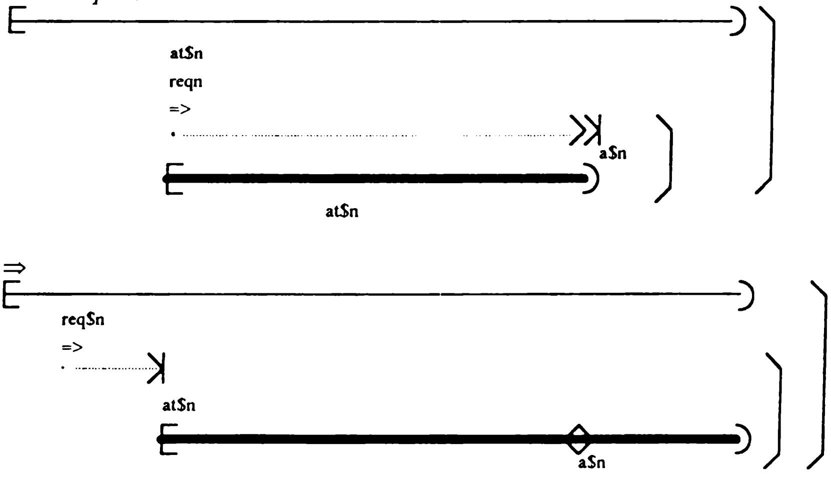
Arrive$n?:



where $\{m, k\} = \{1, 2, 3\} \setminus \{n\}$.

Note: The correctness of the above step can be proved as a lemma.

ServeReqsAt$n:

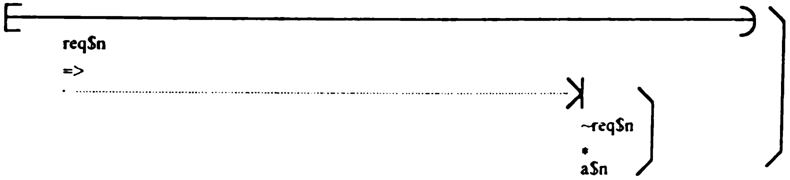


Q.E.D.

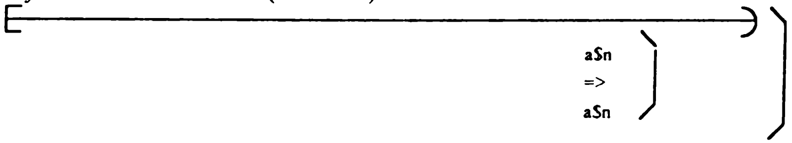
Lemma 1. $WaitService\$n \Rightarrow Arrive\$n$.

Proof.

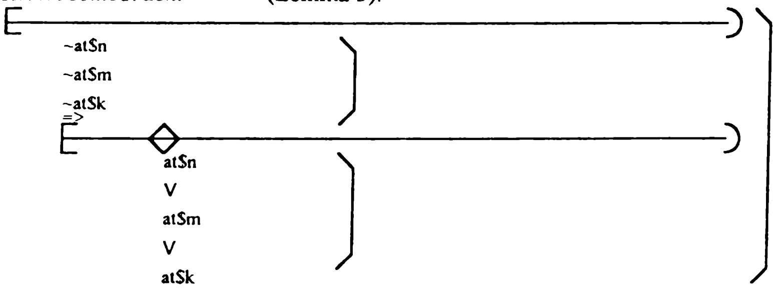
WaitService$n:



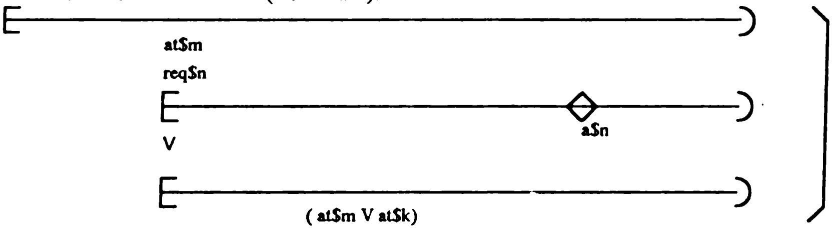
Safe$n: (Lemma 2).



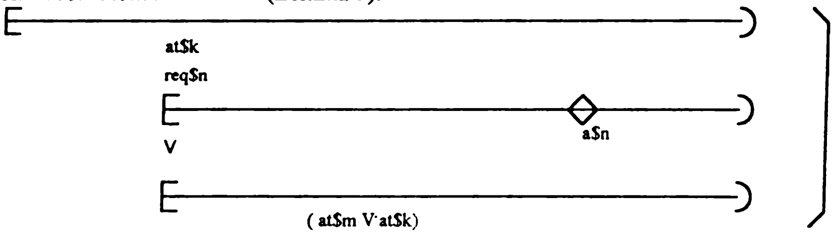
ArriveSomeTrack: (Lemma 3).

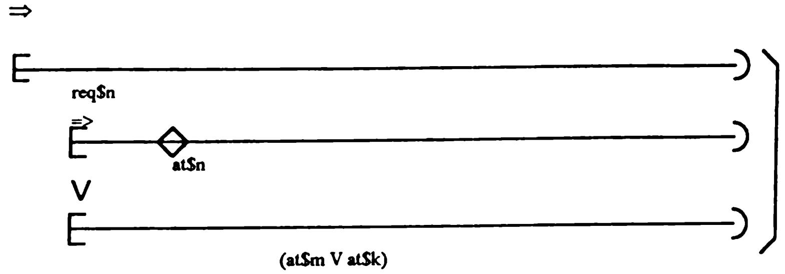


Arrive$n-at$m?: (Lemma 4).



Arrive$n-at$k?. (Lemma 5).





Q.E.D.

The Proof of Lemma 2, Lemma 3, Lemma 4 and Lemma 5 can be constructed in a similar way as the proof of Lemma 1. For example the correctness of Lemma 2 follows from the formulas *Init, SafeAccess$n* and *SafeDepart$n*.

5. Discussion

The above example shows that the specification technique based on graphical interval logic can be applied in the design of the concurrent object-oriented programs as well. We have some experience in specifying concurrent systems using propositional temporal logic. In this paper we have used the GIL and we can state that the complexity of the specification can be really decreased while its reliability is increased [18].

There are several other techniques for specifying synchronizing and scheduling of concurrent systems [10, 19, 20, etc.]. The scheduling predicates introduced in [20] give a powerful mechanism allowing the programmer to schedule the order of execution of operations based on relative arrival times, values of parameters and built-in synchronization counters. The scheduling predicates can be efficiently implemented, but the proof of the derived properties of the specified system is rather complicated. The *control modules* suggested in [19] are the early ancestor of the scheduling predicates. The *event oriented* synchronization technique developed by Laventhal in [10] is an extension of first order predicate logic by so-called events. The result is a mixed system. The implementation of a specification is a hard task and the proof of the properties of the resulted system is very complicated.

References

[1] Concurrent Object-Oriented Programming *(editorial), Comm. of the ACM,* **33** (9) (1990), 125-141.

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

[3] **Tomlinson C. et. al.,** Inheritance and Synchronization with Enabled-Sets, *OOPSLA'89,* 1989, 103-112.

[4] **America P.,** Issues in the Design of a Parallel Object-Oriented Language, *Formal Aspect of Computing,* **1** (1989), 366-411.

[5] **America P.,** Inheritance and Subtyping in a Parallel Object-Oriented Language, *LNCS* **276**, 1987, 234-242.

[6] **Briot J-P. et al.,** Inheritance and Synchronization in Concurrent OOP, *LNCS* **276**, 1987, 32-40.

[7] **Matsouka S. et al.,** Highly Efficient and Encapsulated Re-Use of Synchronization Code in Concurrent Object-Oriented Languages, *OOPSLA'93*, 1993, 109-126.

[8] **Baquero C. et al.,** Concurrency Annotations in C++, *ACM SIGPLAN Notices,* **29** (7) (1994), 61-67.

[9] **Yuen C. K. et al.,** Parallel Multiplication: A Case Study in Parallel Programming, *ACM SIGPLAN Notices,* **29** (3) (1994), 12-17.

[10] **Laventhal M. S.,** Synchronization Specifications for Data Abstractions, *Proc. of IEEE Conference*, 1979, 119-125.

[11] **Kozma L.,** Proving the Correctness of Implementations of Shared Data Abstractions, *LNCS* **137**, 1982, 227-241.

[12] **Kozma L. and Varga L.,** A Methodology for the Development of Shared Object Classes, *Proc. of International Conference on Applied Informatics, Eger, 1993*, 5-14.

[13] **Dillon L.K. et al.,** A Graphical Interval Logic for Specifying Concurrent Systems, *ACM Trans. on Soft. Eng. and Methodology,* **3** (2) (1994), 131-165.

[14] **Rácz É.,** Specifying a Transaction Manager Using Temporal Logic, *Proc. of the Third Symp. on Progr. Lang. and Soft. Tools, Kääriku, Estonia, 1993,* 109-119.

[15] **Wirfs-Brock R.J. and Johnson R. E.,** Surveying Concurrent Research in Object-Oriented Design, *Comm. of the ACM,* **33** (9) (1990), 104-124.

[16] *Concurrent System,* ed. A. Yonezawa, MIT Press, Cambridge, Mass., 1990.

[17] **Kutty G.,***A Graphical Environment for Temporal Reasoning,* Ph.D. dissertation, Department of Electrical and Computer Engineering, University of California, Santa Barbara, USA.

[18] **Pásztorné-Varga K.,** Personal consultation.

[19] **Robert P. and Verjus J.P.,** Toward Autonomous Descriptions of Synchronization Modules, *Proc. of IFIP Congress'77, Toronto, 1977,* 981-986.

[20] **McHale C. et al.,** Scheduling Predicates, *LNCS* **612**, 1991, 177-193.

L. Kozma
Department of General Computer Science
Eötvös Loránd University
VIII. Múzeum krt. 6-8.
H-1088 Budapest, Hungary

É. Rácz
APEH

II. Lajos u. 17-21.
H-1023 Budapest, Hungary