ISSN 0105-8517

SPECIFICATION AND VERIFICATION OF CONCURRENT PROGRAMS

by

Jørgen Staunstrup

DAIMI PB-94
March 1979

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

SPECIFICATION AND VERIFICATION OF CONCURRENT PROGRAMS

Jørgen Staunstrup
Computer Science Department
Aarhus University
Ny Munkegade, DK-8000 Aarhus C
Denmark

March 1979

Abstract

A new technique for specifying and verifying concurrent programs is presented. A specification language for writing abstract specifications of concurrent programs is proposed. Key features of the specification language are means for writing program states as partial histories, for structuring specifications, and for specifying state changes by transition commands. Properties of a program are verified from the abstract specification and not from the implementation.

Keywords and phrases: program specification, verification, concurrent programs, mutual exclusion, fairness, transition commands, abstract data types.

CR categories: 4.2, 4.32, 4.35, 5.24.

This work has been supported by Aarhus University, Denmark and NSF under grant MCS 76-83429; it was carried out while the author was at University of Southern California, Los Angeles.

SPECIFICATION AND VERIFICATION OF CONCURRENT PROGRAMS

Abstract

A new technique for specifying and verifying concurrent programs is presented. A specification language for writing abstract specifications of concurrent programs is proposed. Key features of the specification language are means for writing program states as partial histories, for structuring specifications, and for specifying state changes by transition commands. Properties of a program are verified from the abstract specification and not from the implementation.

This work has been supported by Aarhus University, Denmark and NSF under grant MCS 76-83429; it was carried out while the author was at University of Southern California, Los Angeles.

Introduction

This paper presents a technique for specifying and verifying concurrent programs. The emphasis is on developing precise, yet simple and readable specifications of programs.

The technique is illustrated by a resource scheduler from the Solo operating system (Brinch Hansen 1976). It is verified that the scheduler provides fair and mutually exclusive access to a physical resource. First an abstract specification of the resource scheduler is given. It is then verified that the specification has such properties as fairness and mutual exclusion. When this is done an implementation can be derived. The notation for writing specifications is a separate language from the implementation language (programming language). The implementation language is an engineering product and hence represents a compromise between many different goals (efficiency, clarity, etc.). A specification language on the other hand has one goal: clarity, it is therefore much more suitable for developing programs.

The specification provides an abstract domain close to the problem domain. Doing the verification of key properties in this domain, hopefully, reduces the complexity enough that proofs can be constructed, read, and understood by humans and not require a complex verification system.

It is important to realize that any specification can have errors. Through bitter experience all programmers have realized that this is true for programs. But the use of a more abstract or mathematical notation is certainly in itself no guarantee against errors. Mathematical tools are necessary for a systematic and rigorous analysis, but there is no substitute for simplicity in the specification language and the specifications themselves.

Sections 1-5 introduce and motivate the language for writing specifications. Each construct of the language is introduced by an example demonstrating its main characteristics. The purpose of these sections is to give the reader an intuitive feeling of what the specification language looks like, and why it looks the way it does. The specification language is described in more detail in Staunstrup (1978).

In section 6 it is demonstrated how properties of a specification are verified.

1. The Specification of Abstract Data Types

A program in almost any programming language consists of a data structure (a set of variables) and some statements operating on this data structure. If, however, one takes a closer look at a program, one frequently finds that the data structure consists of many sub-structures each of which is manipulated by a small number of operations. This observation has led to the definition of a concept called an abstract data type: an abstract data type consists of a data structure and a set of operations on the data structure.

The language SIMULA-67 (Dahl 1968) was the first to incorporate an abstract data type as a programming language concept. More recently abstract data types have been included in a number of new programming languages: Concurrent Pascal (Brinch Hansen 1975), Alphard (Wulf 1976), CLU (Liskov 1977), and Euclid (Lampson 1977) to mention a few.

A major part of a typical program written in one of these languages consists of definitions of abstract data types. The Solo operating system (Brinch Hansen 1976) is a good example, here approximately three quarters of the program text is abstract data type declarations. It is therefore natural to let an abstract data type be the unit of specification.

The operations of an abstract data type define all the possible manipulations of the abstract data type and are the only interface to the user of the type. After the application of an operation the abstract data type is in a stable state and stays in this state until another operation is applied. A specification of an abstract data type, therefore, consists of:

i) a definition of all possible operations on it,

ii) an enumeration of all its possible states,

iii) a definition of an initial state.

A specification language is introduced for writing specifications of abstract data types. This chapter introduces and motivates the constructs of the specification language.

2. The Specification of Operations

Each one of a number of concurrent processes needs exclusive access to a resource. The access to the resource is controlled by a resource scheduler. The scheduler is always in one of two states: "free" or "inuse." Initially it is free. A process can then request the resource by calling the operation request, this takes the scheduler to the state inuse. When the process no longer needs the resource it calls the operation release, which takes the scheduler back to the state free.

State changes are specified by transitions. A transition from the state free to the state inuse is written:

```
          free → inuse
```

free is called the pre-state and inuse the post-state of the transition. A transition takes place only when the abstract data type is in the pre-state. The transitions of an abstract data type take place one at a time and are always completed within a finite time.

```
type scheduler
    state status is free / inuse
    initial free

    operation request
    when free → inuse end

    operation release
    if inuse → free end
end scheduler.
```

The specification defines:
- the name of the abstract data type: scheduler,
- the name of the current state: status,
- the two possible distinct states: free and inuse, and
- the possible operations: request and release.

The operations are defined using transition commands (Brinch Hansen and Staunstrup 1977):

```
if transition end
when transition end
```

The transition is performed only if the abstract data type is in the pre-state for the transition. If a process attempts to perform a "when transition command" when the abstract data type is not in the pre-state, it is delayed until the abstract data type is brought into the pre-state by some other process. An attempt to perform an "if transition command" when the abstract data type is not in the pre-state is an exception. An exception is a violation of one of the constraints under which a program is executed and it makes further execution of the program meaningless. The "if transition command" is similar to the guarded if statement proposed by Dijkstra (1975).

The names "free" and "inuse" can be viewed as predicates (with the obvious interpretation that free is true in the state free and false in the state inuse). The transitions are then predicate transformers.

When the transitions are interpreted as predicate transformers, a correct implementation of a transition is a statement (in some programming language) having the same predicate transformer, i.e., when the statement is started in the pre-state, it changes the state to the post-state within a finite time.

There is a close relationship between transition commands and guarded commands (Dijkstra 1975). Indeed, one can view a specification with transitions as an abstract program without statements. By doing this, the concern for what a transition does and how it is done are separated. The former is determined by the specification, the latter by the implementation.

3. The Enumeration of States

The next example is a first in, first out queue. In any state it must be possible to distinguish the number and identity of elements in the queue, as well as their order of arrival. If the capacity of the queue is finite, it is possible to give every possible combination of elements and their order of arrival a distinct name. But this leads to an unmanageably large number of state names. Instead of introducing a separate name for each state: <u>the state of an abstract data type is denoted by one of the possible sequences of operations that leads to it.</u>

For example, the state where the elements b, a, and d have been inserted in the queue, in that order, is denoted by:

```
arrival(b):arrival(a):arrival(d)
```

Note, this may not be the actual sequence of operations leading to that state. Since in general there are many ways of getting to a state, it is simpler to denote a state by one representative sequence of operations.

The symbol E is used for the empty sequence of operations. For any sequence x we have the equalities: x:E = E:x = x.

The state of a queue is a finite sequence of arrivals, the state enumeration is therefore written as follows:

```
E(:arrival(processid)) *
```

where the * indicates zero or more repetitions of the construct within the parentheses.

```
type queue
    state f is E(:arrival(processid))*
    initial E  .
```

The initial state of a queue is the state where no operations, the empty sequence, have been applied.

The purpose of transitions is to define state changes and the conditions under which they occur, so transitions inherently involve state changes. However, parts of the state denotation are not changed, for example, when defining the arrival operation, the sequence of arrivals prior to the current one is not changed. The specification language, therefore, has means for naming such parts of a state denotation which are not changed.

The value of f prior to a transition is called f0. The arrival operation is specified using this notation as follows:

```
operation arrival(i: processid)
if f0 → f0:arrival(i) end
```

In addition, other local names f', f'', ... can be introduced. Consider, for example, the state denotation f':arrival(i), here f' is the sequence preceding arrival(i). Primed variables are always existentially quantified, so the state denotation arrival(i'):f' should be read: there exists a processid i' and a sequence f' such that: f = arrival(i'):f'; in other words there is at least one element in the sequence.

Using this notation the queue is specified as follows:

```
type queue
    state f is E(:arrival(processid))*
    initial E

    operation arrival(i: processid)
    if f0 → f0:arrival(i) end

    operation departure
    if arrival(i'):f' → f' end

    operation head: processid
    if arrival(i'):f' → head = i' end
end queue.
```

The operation arrival(i) takes a queue from any state f0 to the state f0:arrival(i). The operation departure takes any nonempty queue, arrival(i'):f', and removes the first element. The operation head returns the value of the first element of a queue, but does not change the state of the queue.

The length of a sequence f is denoted |f|, i.e.

$$|E| = 0 \quad \text{and}$$
$$|f':\text{arrival}(i)| = |f'| + 1$$

Since the specification of operations implicitly defines all the relevant states the state enumeration (state f is ...) is redundant. In the axiomatic method (Guttag 1975) and (Goguen 1975) a similar redundancy is not required but that approach is also based on the observation that a small subset of the operations can be used to characterize the state space. In the axiomatic method this observation is sometimes stated in what is called a "normal form lemma" (Guttag 1975).

In many programming languages the types of variables must be declared, this makes it easy to detect inconsistencies in the way a variable is used. The enumeration of all states in a specification plays a similar role to the type of a variable and therefore makes it easy to detect if an operation uses a state which is not included in the state space.

4. Structuring the State Space

Frequently it is convenient to break the state into a number of components and view it as the cartesian product of the components. Consider again the resource scheduler presented earlier, and assume that a scheduler keeps additional information about which process has access to the resource. It would be possible to give a distinct name to every possible combination of a process and status of the scheduler. But a much simpler specification is obtained by viewing the state as consisting of two components "holder" and "status." To distinguish between separate components, each is given a name. For each component its domain must be specified.

```
type scheduler
     state status is free / inuse,
           holder is processid
     initial status = free
```

The revised request operation is specified as follows:

```
     operation request(me: processid)
     when status = free → status = inuse, holder = me end
```

In general, a state (or sets of states) is denoted by a list of predicates. The specification of the post-state of request shows an example of this. The ", " (comma) should be read as and, since the post-state must satisfy both predicates.

The release operation is written as follows:

```
operation release
if status = inuse → status = free end
end
```

5. Hierarchical Specifications

Frequently, the properties of a data type are too complex to be given as one specification. The data type should then be decomposed into several specifications, each describing some aspect of the type. The specification can then be studied one component at a time.

A hierarchical specification of two components t1 and t2 where the specification of t1 uses the specification of t2 is written as follows:

```
type t2
    state s is u1 / u2 ...
    initial ...

    operation p1 ...
    .
    .
    operation p2 ...
end t2.
```

In the specification of t1 the declaration

```
state a is t2
```

indicates that the name a is used to reference a value of the type t2. The predicates q0, q1, and q2 can, for example, use a.

```
type t1
      state a is t2
      initial q0

      operation r1
      when q1 → q2 end
      .
      .
      .
end t1.
```

The following example is taken from the Solo system (Brinch Hansen 1976). The specification of a resource scheduler is decomposed into two abstract types. One, called resource, specifies that the resource can be used by at most one process at a time (mutual exclusion). When several processes want access to the resource at the same time, all but one of them must wait. The queueing of the delayed processes is specified by the other abstract data type, called fifo.

5.1 Fifo

The fifo is a first in, first out queue of process identities (processid). The capacity of the queue is defined by a constant limit.

```
type fifo
      state f is E(:arrival(processid))*,
      initial f = E

      operation arrival(i: processid)
      if |f| < limit → f = f0:arrival(i) end

      operation departure
      if f = arrival(i'):f' → f = f' end

      operation head: processid
      if f = arrival(i'):f' → head = i' end

end fifo.
```

5. 2 Resource

A resource scheduler gives processes exclusive access to a computer resource. It uses "fifo" to control the order in which processes get access to the resource.

All processes using a resource r must perform the following sequence of operations: ‡)

```
r.request(me);
r.grant(me);
use resource;
r.release
```

where me is the identity of the calling process.

```
type resource
    state q is fifo,
          free is boolean
    initial q.initial, free

    operation request(i: processid)
    if q = q0 → q = q0.arrival(i) end

    operation grant(i: processid)
    when q.head = i, free → q = q0.departure not free, end

    operation release
    if not free → free end

end resource.
```

The initial state of an abstract data type t is denoted t. initial. This is used in the specification of the initial state of a resource, where q. initial is used to denote the initial state of the fifo q.

‡) The resource scheduler in the Solo system combines the two operations request and grant in one operation. For a further discussion of this point you are referred to Staunstrup (1978).

6. Verifying Properties of a Specification

The specification of a program is a precise formulation of the programmer's requirements to the program. The specification is the formal model of the programmer's informal understanding of the program. When writing the specification, the programmer tries to include all the desired properties of the program; but it is in general impossible to prove that the specification is correct, i.e., that it captures all the desired properties.

When the specification is completed he is on firm ground and can verify whether or not it has a certain property. For example, when the specification of the resource scheduler is written, it can be verified that it has such properties as mutual exclusion and fairness.

It is, however, not possible to foresee all the uses of a program (or an abstract data type) when it is specified. Therefore, it should be possible to keep the specification and use it for verifying properties also after the design is finished. This philosophy is illustrated below, where it is shown that the resource scheduler specified in section 5 provides mutually exclusive and fair access to the resource. Let r be an instance of the resource scheduler. Processes using the resource scheduled by r must follow the pattern:

```
r.request(me);
r.grant(me);
critical region;
r.release;
```

Furthermore, assume that all processes are in their critical region for a finite time only.

6.1 Mutual Exclusion

The following approach was originally proposed in Brinch Hansen and Staunstrup (1978).

The resource scheduler provides exclusive access to the resource if no process is able to complete a grant operation while another process is in its critical region. Let inside(p) denote the state of the resource where process p is in its critical region, and let pre-grant(i) denote the pre-state for grant(i). Mutual exclusion is guaranteed if:

<u>for</u> <u>all</u> i <u>in</u> processid:($i \neq p$)
    (inside(p) $\Rightarrow$ <u>not</u> pre-grant(i))

When <u>not</u> pre-grant(i) holds, process i will not be able to complete the grant operation and is therefore prohibited from entering the critical region.

When process p is in its critical region, it has completed a grant operation. At this point a number of requests can have been made:

inside(p) $\equiv$ grant(p).request(i1).request(i2). ... .request(in).

From the specification of grant and request it follows that:

inside(p) $\Rightarrow$ <u>not</u> free,

because grant takes the resource to a state where <u>not</u> free holds and request does not change free. But since

pre-grant(i) $\equiv$ (q.head = i) <u>and</u> free

it follows that:

inside(p) $\Rightarrow$ <u>not</u> pre-grant(i).

So the resource scheduler guarantees mutual exclusion.

<u>6.2 Fairness</u>

The scheduler is <u>fair</u> if all processes requesting the resource are allowed to enter their critical region within a finite time.

The state of the resource just after a request by process p is: r!.request(p). The process p will enter its region only if the resource is in the pre-state for grant(p): (q.head = p) <u>and</u> free. Fairness is therefore guaranteed if it can be shown that:

$$r'.\text{request}(p) \overset{*}{\rightarrow} (q.\text{head} = p) \ \underline{\text{and}}\ \text{free}$$

that is, <u>a finite number of transitions</u> will be made which will transfer the resource from the state $r'.\text{request}(p)$ to the state $(q.\text{head} = p)$ <u>and</u> free. (Each transition takes a finite time only, so a finite number of transitions also take a finite time.)

This property is proved by induction on the length of r' (the length of the queue when process p joins it).

First, assume that the length of r' is 0. The queue is then empty so: $q.f = E$. It follows from the specification of request that:

$$\{q.f = E\}$$
$$r.\text{request}(p);$$
$$\{q.f = \text{arrival}(p)\}$$

and from the specification of head that:

$$q.f = \text{arrival}(p) \Rightarrow q.\text{head} = p.$$

So, when the length of r' is 0, we have:

$$r'.\text{request}(p) \rightarrow q.\text{head} = p.$$

Another process might be in its critical region, but it was assumed that it would leave the region within a finite time. From the specification of release it follows that:

$$\{q.\text{head} = p\}$$
$$r.\text{release};$$
$$\{(q.\text{head} = p)\ \underline{\text{and}}\ \text{free}\}.$$

This is the base step of the induction.

Next, assume that the property has been shown for all r' with a length less than k $(k > 0)$, and assume that the length of r' is k. From the specification of request, it is seen that:

```
{q.f = arrival(i1):arrival(i2): ... .arrival(ik)}
r.request(p);
{q.f = arrival(i1):arrival(i2): ... :arrival(ik):arrival(p)}.
```

From the assumption about the behavior of all processes, we know that process i1 will complete a grant and therefore a departure within a finite time:

```
{q.f = arrival(i1):arrival(i2): ... :arrival(ik):arrival(p)}
r.grant(i1);
{q.f = arrival(i2): ... :arrival(ik):arrival(p)}.
```

From the induction hypothesis we get:

q.f = arrival(i2): ... :arrival(ik):arrival(p)

$\downarrow$ *

(q.head = p) <u>and</u> free.

Requests arriving after request(p) can be ignored. This follows from the following property of an instance q of fifo.

Let a,a1,a2,...,an (n ≥ 0) be arbitrary, possibly empty, sequences of arrivals, we then have:

```
{q.f = arrival(i1):arrival(i2): ... :arrival(in):arrival(p):a}
q.a1; q.departure;
q.a2; q.departure;
.
.
.
q.an; q.departure;
{q.head = f}
```

(It is assumed that the capacity of q is never exceeded.) The proof of this can be found in Staunstrup (1978). This proves the induction step.

Conclusion

Traditionally program verification is done by formulating precise requirements to a program and relating them to the program text, for example, in the form of assertions merged with the text of the program. The main contribution of this work is to show that there are viable alternatives to that approach. Instead of writing the requirements concretely in the form of assertions about program variables etc., an abstract specification of the program is given. The specification is not as detailed as the program, so it is more readable. Verifying that it has the desired properties is therefore simple. Furthermore, different implementations of the same specification all have these properties, no additional proofs are necessary.

The key features of the specification language are:

- the characterization of program states as partial histories of operations represented as sets, sequences, and cartesian products,
- the structuring of a specification into a hierarchy of simple specifications, and,
- the non-deterministic transition commands which are used for specifying both sequential and concurrent programs.

The specification language has been tried on a large number of examples. These have all yielded simple and readable specifications. Based on this evidence, I am convinced that the specification language, despite its simplicity, is a helpful and powerful tool for program construction. Future experience might, however, reveal extensions of the specification language.

This paper has only presented some of the constructs of the specification language. In Staunstrup (1978) the specification language is discussed in more detail. In that work techniques for constructing correct implementations are also presented, and a major part of the Solo operating system (Brinch Hansen 1976) is specified and implemented.

Acknowledgement

I wish to thank Per Brinch Hansen and John Guttag for many stimulating discussions in various stages of this work.

References

Brinch Hansen, P., The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975), 199-207.

Brinch Hansen, P., The Solo operating system, Software Practice and Experience 6, 2 (April - June 1976), 144-205.

Brinch Hansen, P., and Staunstrup, J., Specification and implementation of mutual exclusion. IEEE Transactions on Software Engineering 4, 5 (September 1978), 365-71.

Dahl, O.-J., Myhrhaug, B., and Nygaard, K., Simula 67 - common base language, Norsk Regnesentral, Oslo, Norway, May 1968.

Dijkstra, E.W., Guarded commands, nondeterminacy, and formal derivation of programs. Comm. ACM 18, 8 (August 1975), 453-57.

Goguen, J.A., Thatcher, E.G., and Wright, J.B., Abstract data types as initial algebras and correctness of data representation. Proceedings of Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975.

Guttag, J.V., The specification and application to programming of abstract data types. Ph.D. Thesis, University of Toronto, Department of Computer Science, 1975.

Guttag, J.V., Horowitz, E., and Musser, D.R., Abstract data types and software validation. Comm. ACM 21, 12 (December 1978), 1048-64.

Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J., Report on the programming language Euclid. SIGPLAN Notices 12, 2 (February 1977).

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., Abstraction mechanisms in CLU. Comm. ACM 20, 8 (August 1977), 564-76.

Staunstrup, J., Specification, Verification, and Implementation of Concurrent Programs. Ph.D. dissertation, Computer Science Department, University of Southern California, Los Angeles, Ca., May 1978.

Wulf, W.A., London, R.L., and Shaw, M., Abstraction and verification in Alphard: introduction to language and methodology. Carnegie-Mellon University and USC Information Sciences Institute Technical Reports, 1976.