Abstract: The algebra of Communicating Sequential Processes provides a facility to formal specify systems by describing their behavior. To be able do denote these systems more precisely it is useful to introduce state variables. Because the CSP algebra deals with concurrent processes, the state variables are shared between those processes. This contribution describes techniques how to represent shared variables in CSP utilizing common features of CSP without any extension to the algebra.

Keywords: CSP, process, concurrency, variable, shared variable, critical section, formal specification, communication channel

1. Introduction

At present day, widely used hardware and software systems occur in applications where failure is unacceptable. These systems surround us every day and we often do not realize how much a failure can impact our lives. Airport control systems, bank systems, medical instruments, and other examples, a small list of most critical applications where an error can cause human deaths. Clearly, the need for reliable systems is critical. As the usage of such systems grows, so too does the need for their correctness. Moreover, with successful expansion of the Internet and embedded systems in cars, airplanes, etc. should these systems be even more reliable, [1]. Verification proves the system’s reliability.

The first step in system verification lies in specifying which properties should the system fit. For example concurrent system should fit the property that it never reach deadlock. After defining the system properties, the second step represents construction of the system’s formal model. The specification should include only that properties which must the system fit to be reliable. Construction of such specification represents a difficult topic. The more detailed specification of the critical parts the better. Therefore, it is useful to extend the specification by state variables. Because the CSP algebra deals with concurrent processes, the state variables are shared between those processes.

This contribution introduces technique how to specify shared variables using common CSP resources.

In contrast to state variables described in CSP the shared variables are accessible from distinct processes. The specification of shared variables utilizes communication channels to express particular variables of different types. This paper also describes special processes which deals with shared variables and shows how to express their values.

First part of this contribution describes the relevant subset of CSP algebra used in the introduced technique, the second part describes how to specify shared variables utilizing communication channels and the third part denotes processes which deals with shared variables.

2. Communicating Sequential Processes subset

Understanding, designing and building concurrent systems represent a major challenge for computer science. The involved complications differ from the sequential programming problems, therefore the concurrent systems requires systematic approaches.

Concurrent systems are all around us. They consist of independent, but communicating components. The familiar examples include:
- the network of bank cash machines
- the internet
- the telephone system
the components of a PC

The algebra of CSP provides a possibility for concurrent systems to be modeled in more elementary and abstract way. It is supported by particular software tools which offer system analysis and verification.

CSP describes processes – objects which exist independent on each other, but may communicate. During their lifetime, processes can perform various actions or events. These events represent the visible part of modeled processes. For example, when describing a simple vending machine, two events may be interesting:

1. **coin** – represents insertion of a coin
2. **choc** – represents the appearance of a chocolate

The set of events used by the process to represent its behavior is called alphabet or interface. During the process activity the events in the interface may occur once, many times, or not at all. Which events should be included in the interface depends on aspects of process behavior which are interesting. For example, when specifying a lecture and interesting just for the beginning and the end of the lecture then the interface of the process consists of two events – *begin* and *end*.

The simplest possible process behavior stands for do nothing written as *STOP*. Whenever the behavior of a system reaches this process then deadlock occurred. Nontrivial processes are written by means of prefixing operator which allows events to occur in sequence. So, when \( P \) is a process and \( a \) an event then \( a \rightarrow P \) represents a process which performs the event \( a \) and then behaves like process \( P \). Expressions of the type \( P \rightarrow Q \) or \( a \rightarrow b \) are not allowed. The prefix operator defines the relation between events and processes only (for example: \( a \rightarrow P, \ a \rightarrow b \rightarrow P, \) etc.).

Except *STOP* another predefined process exists in CSP – *SKIP*. Like *STOP*, it does nothing but ends correctly. Therefore, the *SKIP* process indicates the correct termination of a process.

Utilizing predefined processes and the prefix operator, only finite processes can be created. But often have to be specified processes that run forever. To achieve this goal recursion is included. For example, specification of a clock using an event *tick* describes the following process: \( \text{CLOCK} = \text{tick} \rightarrow \text{CLOCK}. \) The process *CLOCK* performs the event *tick* repeatedly.

Specified processes often don’t just perform single sequence of events but may have alternative behavior caused by their environment, for example. So, if \( P \) and \( Q \) are processes and \( x \) and \( y \) are distinct events, then the process \( x \rightarrow P \mid y \rightarrow Q \) performs either the event \( x \) and then behaves like process \( P \) or performs the event \( y \) and then behaves like process \( Q \).

Modeled processes usually don’t appear isolated but interfere with other process, for example with the process’ environment. Mutual interaction between two or more processes means that these processes perform common events simultaneously. The alphabet of events specifies on which events the processes synchronize. For example, when describing the vending machine again, the new process representing the customer interacts with the machine. Example 1 describes these interacting processes.

Example 1:

\[
\text{MACHINE} = \text{coin} \rightarrow (\text{choc} \rightarrow \text{MACHINE} \\
| \text{coffee} \rightarrow \text{MACHINE})
\]

\[
\text{CUSTOMER} = \text{coin} \rightarrow \text{choc} \rightarrow \text{SKIP}
\]

\[
\text{SYSTEM} = \text{MACHINE} \downarrow \text{CUSTOMER}
\]

\[
A = \{\text{coin}, \text{choc}, \text{coffee}\}
\]

So far the utilized events were considered regardless of whether they represent inputs or outputs. However, separated notation for input and output may be useful for some cases. For this purpose a special event in the form \( c.v \) is defined, where \( c \) stands for the communication channel name and \( v \) stands for the message value send through the channel. Each channel has a type which simply represents the set of events which can be transmitted among the channel. The event \( c.v \) represents a general notation for communication among channels. Therefore, to support sending and receiving of messages, two new operators are defined: process \( c!v \rightarrow P \) sends a message \( v \) among the channel \( c \) and then behaves like \( P \), process \( c?v : T \rightarrow P(x) \) receives the message \( x \) of the type \( T \) and then behaves like \( P(x) \). Until a message of the specified type appears on the input the receiving process waits.

When describing concurrent systems which have some processes with similar behavior it is useful to specify the process’ behavior only once and then use this specification for particular processes. To achieve this, each process is labeled by a different name and each event from the labeled process is also labeled by this name. A labeled event is a pair \( l.x \) where \( l \) is a label and \( x \) is a symbol standing for the event. Then a process \( P \) labeled by \( l \) is denoted by \( l.P \)

It engages in the event \( l.x \) whenever \( P \) would have engaged in \( x \). The alphabets of labeled processes are
distinct, so labeled processes can interact with each other on particular events.

The complete algebra of CSP provides much more notations but for this contribution purposes the presented subset fit the requirements. The complete algebra of Communicating Sequential Processes and its strict description is given in [2], while [3] presents more simplified version.

3. Failures Divergence Refinement

Failures Divergence Refinement (FDR) facilitates verification of many finite system properties and analysis of systems which fail the test. It stems from the Communicating Sequential Processes theory and utilizes refinement theory which provides huge range of correctness requirements including the absence of deadlock and livelock. FDR includes also requirements for general safety and liveness properties.

FDR provides understandable and usable capabilities and extensive debugging facilities to support system development. Therefore, FDR is suitable for verification of systems with complex behavior. When an error occurs, FDR describes the state that lead to the failure as well as the sequence of events that engaged in this state. At present day FDR can analyze extremely large state-space (for example $7^{2^{1024}}$) within few minutes on common desktop PCs.

FDR was specifically developed for analysis and verification of industrial applications. It was successfully applied on VLSI circuits, embedded systems, etc. Another major group of applications involve using FDR to check communications and communication protocols, specifically to detect security holes by authentication key exchange.

4. Shared variables in CSP

The algebra of Communicating Sequential Processes includes state variables, which are variables locked in one process and these variables are not accessible from other processes. On the other hand, in concurrent systems, particular processes communicate with each other. There are few ways how to perform communication between processes. One of these approaches utilizes shared variables which are accessible from all processes. This chapter introduces a technique how to describe shared variables using CSP and how to deal with invited problems with variables shared among variety of processes.

4.1. Basic idea of Shared variables

The basic idea of shared variables utilizes communication channels to represent the variables' values. The name of the communication channel stands for the name of the shared variable and the value sent among the channel corresponds to the value stored in the shared variable. Then, when describing a process which writes a value to the variable, the writing process contains an event which sends the value of the shared variable to the channel. Likewise writing, when a process reads a value from a shared variable the reading process contains event which reads a message from the channel. For example, when describing a variable “store” the writing process looks follows:

writing to the shared variable: \( \text{store!value} \)

reading from the variable: \( \text{store?value} \)

This approach to shared variables is quite simple and easy to implement, but this specification does not correspond to real concurrent processes. Because communication channel works in synchronous mode, a process which sends a value among the channel waits until other process reads this value from the channel. In real concurrent systems a process stores the value in the variable and then continues in his behavior without synchronization. Therefore another approach should be invited to preserve correspondence between real system and its formal specification.

4.2. Shared variable as a special process

The in previous paragraphs presented approach needs to be extended so that processes don't have to synchronize when accessing shared variable. This goal can be achieved by introducing a special process which represents the shared variable. The process utilizes two separate channels, one to read the value from the writing process, other to send the value to the reading process. Specification of such process look follows:

\[ \text{VAR} = \text{left?x} \rightarrow \text{right!x} \rightarrow \text{VAR} \]

Then, labeling of processes distinguishes between particular variables. The used label denotes the name of the shared variable and all the particular events of the \( \text{VAR} \) process are distinguished by the prefixing label, too.
The specification of the “store” variable is denoted as follows:

Variable definition: \( store : VAR \)
Writing process: \( store.left!value \)
Reading process: \( store.right?value \)

The reading and writing process should be put in parallel with the \( VAR \) process to satisfy the correctness of the specification.

This specification of shared variable comes nearer to the real concurrent system, but it still has some disadvantages. The main disability is that such specified variables don’t allow multiple storages of a value in the variable. The \( VAR \) process reads a value from the writing process and then waits until a reading process reads the value from the \( right \) channel. In between those two events the \( VAR \) process is unable to read next value. Therefore, the \( VAR \) process specification has to be extended by alternative behavior. The alternate behavior provides the ability to multiple write values to the shared variable. The new description of the \( VAR \) process is denoted bellow:

\[
VAR = left?value \rightarrow VAR(value) \\
\mid right!value \rightarrow VAR \\
\]

This specification of shared variable is more precise and near to the real shared variable, but it still has disadvantages. Processes can multiply write values to this variable, but it can be only once read. After this operation, the corresponding communication channel loses the value of the variable and other processes can not access it again.

Solution of this problem lies in adding one more process to each shared variable. This new process stores temporally the value of the variable and immediately after the reading event successfully ends it stores the value again in the shared variable. The final description of shared variable is denoted bellow; the two processes \( VAR \) and \( VAR2 \) communicate among the same channels and have the same labels:

\[
VAR = left?value \rightarrow VAR(value) \\
\mid right!value \rightarrow right2!value \rightarrow VAR \\
VAR2 = right2?val \rightarrow lock \rightarrow left!val \rightarrow unlock \rightarrow VAR2 \\
\]

Such shared variable specification corresponds to a real concurrent system’s shared variables. All processes can access such variable, write a value to the variable and read the value from the variable. By achieving this specification, common problems with variables shared among particular processes arise and have to be solved.

4.3. The Critical Section problem

When handling with variables shared among more than two processes a problem called critical section arises. It is a problem when two or more processes simultaneously access the same shared variable. Then the results of operations over such variables are nondeterministic and can not be successfully verified.

The solution of the critical section problem was presented in many ways, for example semaphores, monitors, etc. In this contribution we will use the semaphore approach which uses two special operations \( P \) and \( V \), one for entering the critical section, other for leaving the critical section. For this paper purposes two special events represents the two operations. Therefore, the \( VAR \) process specification has to be extended by this two events and the specification of the shared variable looks follows:

\[
VAR = lock \rightarrow left?value \rightarrow unlock \rightarrow VAR(value) \\
\mid right!value \rightarrow right2!value \rightarrow VAR \\
VAR2 = right2?val \rightarrow lock \rightarrow left!val \rightarrow unlock \rightarrow VAR2 \\
\]

The events \( lock \) and \( unlock \) represents the two semaphore operations. These events are performed only when processes write a value to the shared variable, not by reading of the value. Consequently, only one process can write to the variable, but all processes can simultaneously read the value. If total mutual exclusion of processes were required when accessing the shared variable, the \( lock \) and \( unlock \) events needs to be included in the reading part of the specification, too.

5. Dealing with shared variables

In concurrent programming, the processes don’t only read and write values from and to shared variables, but also deals with the variables and often decide its behavior depending on the shared variable’s value. For example a cycle or a simple condition takes two variables and whether the condition holds the loop or the statement behind the condition is performed. To specify handling with shared variables a special process has to be invited. The process compares two variables and then
performs the corresponding event. Specification of such process is denoted bellow:

\[
\text{COMPARE} \rightarrow \text{compare}.x.y \rightarrow \text{LOOP}(x, y) \\
\text{LOOP}(x, y) \rightarrow \text{compare}.0.0 \rightarrow \text{equal} \rightarrow \text{COMPARE} \\
| \text{compare}.x.0 \rightarrow \text{greater} \rightarrow \text{COMPARE} \\
| \text{compare}.0.y \rightarrow \text{lower} \rightarrow \text{COMPARE} \\
| \text{loop} \rightarrow \text{LOOP}(x-1, y-1)
\]

The first process synchronizes on the event \text{compare}.x.y with systems processes which will compare two variables, the second process performs a loop which counts down the value of both variables. If one or both of the variables reaches zero, then the loop ends and the \text{LOOP} process performs corresponding event which evaluates the comparison. Those evaluating events again synchronize with the system’s processes. Example 2. denotes a simple system with condition.

Example 2:
If \((x < 5)\) \(x = x + 1\);
\text{SYSTEM} = x.right!val \rightarrow \text{compare}.val.5 \rightarrow \text{EVALUATE} \\
\text{EVALUATE} = \text{lowe} \rightarrow x.right?val \rightarrow x.loc \rightarrow x.left!(val+1) \\
\rightarrow x.unlock \rightarrow \text{SKIP} \\
| \text{equal} \rightarrow \text{SKIP} \\
| \text{greater} \rightarrow \text{SKIP}

6. Types of shared variables

Up to this time, when we spoke about shared variable we newer mind the type of the variable. All the processes denoted before handled with variables as they were integers, but we often need to specify variables which have other type, i.e. fields, strings, real numbers, etc. Therefore, a solution which will be able to deal with these types of variables has to be included, too.

The simplest solution of this problem lies in retyping all these variables to integers and then use the before denoted processes, without change. The following paragraphs describe the retyping technique for the mainly used types of variables.

Real numbers: real numbers can be transformed to integers by multiplying the value by multiple of 10. When comparing two numbers, both have to be multiplied by the same number to comprehend the system’s correctness.

Fields: when dealing with fields, all particular field elements have the same type. So, when assuming that each type can be retyped to integers, it is necessary to define only fields of integers. This is accomplished by labeling, because labeling can be used recursively. For example field “store[20]” is denoted follows:

\(i : \text{store} : \text{VAR}; i = 1, 2, ..., 20\)

The field element is then accessed thorough the element’s index, for example value of “store[3]” is accessed follows:

\(3.\text{store.right?value}\)

Strings: a string is a field of characters. Because we already know how to specify fields of integers, so we only need to transform a character to an integer. Because all characters in computers occur in tables where each character has an unique ordinary number, these numbers will represent the characters equally. In other words, the string is a field of ordinary numbers of particular characters in the string.

7. Conclusions

The need for reliable systems grows from year to year. Moreover, with the terrific expansion of embedded systems and Internet have these systems to be even more reliable. Therefore, the system needs to be verified. First step in system verification lies in specifying the requirements on the system, second step represents creation of system’s formal model. To be suitable for verification, the model has to fit system requirements which should be satisfied for system correctness. On the other hand, the model should abstract those properties which don’t influence the system correctness and complicate the verification. Although the verification process of formal models represents a long-time procedure, it is applied on more and more systems because finding and eliminating of consequent errors stand for high costs.

This contribution introduces shared variables as a medium to achieve more precise formal specifications of concurrent systems. The shared variables utilize common facilities of CSP so that the algebra of Communicating Sequential Processes doesn’t need to be extended. A special set of two processes represents each shared variable and utilizes communication channels to store the value of the variable. The paper discusses multiple possibilities how to represent the shared variable as a process including the specification which corresponds to real concurrent system’s shared variable. The second part
of this contribution describes dealing with shared variables. Again, a special process denotes the facility of comparing shared variables. The last part contains instructions how to represent other types of variables as integers and so use only the described notations for multiple variable types.

References: