The sire language defintion

James Hanlon

Last revised July 18, 2014

Contents

1 Introduction

Sire[1] is a communicating-process programming language designed for highly-parallel execution. It provides facilities for sharing and abstraction that can be used to build distributed parallel subroutines and shared-access data structures. Its design is based on the occam programming language and, in the occam philosophy, includes the smallest set of features necessary to this.

Sire is capable of being compiled using simple techniques to produce efficient programs, allowing the programmer to directly exploit the UPA.

This document presents the definition of the sire. It introduces the language incrementally, with each section building on the last.

[1]The word *sire* is a verb meaning to create or bear. The choice of this for the name of the language relates to the way in which a sire program executes, by allocating and deallocating processors dynamically.

2 The model of computation

A sire program consists of a collection of *processes*. Each process performs a sequence of commands, as well as being able to create additional processes. Processes can communicate with each other by using point-to-point message passing *channels*, which consist of a connected pair of *channel ends* local to either process. Communication also causes processes to synchronise; a sending process waits until the receiver is ready to receive a message. A special type of process called a *server* provides a means of abstraction and forms the basis of a parallel subroutine mechanism.

A sire program is executed by a collection of one or more *processors*. Each processor has a small private memory and can execute a number of *processes* simultaneously. The execution begins on a single processor and the computation progresses in *time* and *space* as processors are allocated and deallocated dynamically.

3 Notation

Since sire builds on occam, it is prudent to present the language in a similar way to the original occam specifications, as its successors have also done [**?**, **?**, **?**]. The presentation uses a modified version of the *Backus-Naur Form* (BNF) to specify the syntax of the language. In the BNF, a *production rule* defines a new *symbol* to consist of a choice of one or more sequences of other symbols, which are defined by other rules or are *terminal* and therefore elements occurring in the program. Taking part of the command syntax as an example, a production of the form

$$\textit{command} = \langle\text{input}\rangle \quad | \quad \langle\text{output}\rangle$$

specifies the symbol ⟨process⟩ to be the symbol ⟨input⟩ or ⟨output⟩. This can also be written with separate rules in the equivalent definition

$$\begin{aligned}\textit{command} &= \langle\text{input}\rangle \\ \textit{command} &= \langle\text{output}\rangle\end{aligned}$$

in order to incrementally introduce elements of the syntax. The symbols ⟨input⟩ and ⟨output⟩ are specified with the rules

$$\begin{aligned}\textit{input} &= \langle\text{chanend}\rangle \ \textbf{?} \ \langle\text{variable}\rangle \\ \textit{output} &= \langle\text{chanend}\rangle \ \textbf{!} \ \langle\text{expression}\rangle\end{aligned}$$

The emboldened symbols '**?**' and '**!**' are terminals and ⟨chanend⟩, ⟨variable⟩ and ⟨expression⟩ are defined in terms of one or more additional production rules.

The notation $\{_c X\}$ specifies a list of c or more Xs and is equivalent to $X_1 X_2 \cdots X_c \cdots$, and $\{_c \oplus X\}$ specifies a list of c or more Xs separated by $\oplus$ and is equivalent to $X_1 \oplus X_2 \oplus \cdots \oplus X_c \oplus \cdots$.

As each part of the language is introduced, fragments of the BNF are given and the semantics of the syntax explained. The semantics of a small subset of the language are given informally, which allows the remaining portion to be defined algebraically, building on the subset. This approach is attractive since it provides clear semantics and allows simple transformations between different constructs.

Where examples of sire syntax are given the notation $[\![\cdot]\!]$ is used to distinguish sire syntax from mathematical notations.

4 Primitive commands

$$
\begin{aligned}
\textit{command} \;=\; & \langle \text{assignment} \rangle \\
\mid\; & \langle \text{input} \rangle \\
\mid\; & \langle \text{output} \rangle \\
\mid\; & \langle \text{connect} \rangle \\
\mid\; & \langle \text{skip} \rangle \\
\mid\; & \langle \text{stop} \rangle
\end{aligned}
$$

A program is built from commands and the simplest of these are *primitive* commands, which are *assignment*, *connect*, *input*, *output*, *skip* and *stop*.

4.1 Assignment

$$\textit{assignment} = \langle \text{variable} \rangle \;\texttt{:=}\; \langle \text{expression} \rangle$$

An *assignment command* evaluates the right hand side expression and changes the value of the variable to the result.

4.2 Input and output

$$
\begin{aligned}
\textit{input} &= \langle \text{chanend} \rangle \;\texttt{?}\; \langle \text{variable} \rangle \\
\textit{output} &= \langle \text{chanend} \rangle \;\texttt{!}\; \langle \text{expression} \rangle
\end{aligned}
$$

Values are passed between processes using *bidirectional communication channels*. These consist of two *channel ends* that are associated with either process communicating on a channel. Communication is *synchronised*, which means that an outputting process must wait for the inputting process to be ready before any data is sent. This prevents any data from being lost due to programming errors and precludes the use of buffering.

An *input command* receives a value from a channel and assigns it to a variable and an *output command* evaluates an expression and sends the result on a channel. Matching pairs of input and output are a distributed form of assignment.

4.3 Connect

$$
\begin{aligned}
\textit{connect} &= \texttt{connect}\; \langle \text{chanend} \rangle \;\texttt{to}\; \langle \text{chanend} \rangle \\
\textit{chanend} &= \langle \text{element} \rangle
\end{aligned}
$$

A *connection* between two channel ends is established with a matching pair of *connect* commands. Each connect command connects a *local* channel end to the corresponding *remote*

channel end. The commands terminate when the connection has been established. The channel end itself is an element that specifies a name or field (see §10, p. 21).

4.4 Skip and stop

$$\begin{aligned} \mathit{skip} &= \texttt{skip} \\ \mathit{stop} &= \texttt{stop} \end{aligned}$$

The command **skip** does nothing and terminates, the command **stop** never terminates, causing the process to execute no more actions. These commands are used primarily to explain the behaviour of other constructs in the language.

5 Structured commands

$$
\begin{aligned}
\textit{command} &= \langle \text{alternation} \rangle \\
&\mid \langle \text{conditional} \rangle \\
&\mid \langle \text{loop} \rangle
\end{aligned}
$$

The structured commands *conditional*, *alternation* and *loop* are control constructs that are used to combine commands.

5.1 Alternation

$$
\begin{aligned}
\textit{alternation} &= \texttt{alt \{} \{_0 \texttt{ | } \langle \text{alternative} \rangle \} \texttt{ \}} \\
\textit{alternative} &= \langle \text{guarded-alternative} \rangle \\
&\mid \langle \text{alternation} \rangle \\
\textit{guarded-alternative} &= \langle \text{guard} \rangle \texttt{ : } \langle \text{command} \rangle \\
\textit{guard} &= \langle \text{input} \rangle \\
&\mid \langle \text{expression} \rangle \texttt{ \& } \langle \text{input} \rangle \\
&\mid \langle \text{expression} \rangle \texttt{ \& } \langle \text{skip} \rangle
\end{aligned}
$$

An *alternation command* is used to deal with non-determinism by allowing a process to wait for inputs on a number of different channels. It consists of a list of *guarded command alternatives*. A guard consists of an expression and an input on a channel or **skip** in the place of the input if none is required. When the value of the expression is **true**, the guard behaves like the input or **skip**, otherwise it behaves like **stop** and does not proceed. Conceptually, the guard expression *enables* or *disables* an alternative.

When an alternation command is performed, it behaves like one of the alternatives that can proceed. Without any alternatives, it behaves like **stop**. An alternative that is itself an alternation is ready when one of its alternatives is ready.

5.2 Conditional

$$
\begin{aligned}
\textit{conditional} &= \texttt{if \{} \{_0 \texttt{ | } \langle \text{choice} \rangle \texttt{ \}} \\
&\mid \texttt{if } \langle \text{expression} \rangle \texttt{ then } \langle \text{command} \rangle \texttt{ else } \langle \text{command} \rangle \\
\textit{choice} &= \langle \text{guarded-choice} \rangle \\
&\mid \langle \text{conditional} \rangle \\
&\mid \langle \text{specification} \rangle \texttt{ : } \langle \text{choice} \rangle \\
\textit{guarded-choice} &= \langle \text{expression} \rangle \texttt{ : } \langle \text{command} \rangle
\end{aligned}
$$

A *conditional command* consists of a list of expressions enumerating a set of choices, with a command associated with of each. When a conditional command is performed, each choice is

tested in sequence and the command behaves like the first choice that evaluates **true**. Without any choices, it behaves like **skip**.

An alternative *if-then-else* form of the conditional command combines two choices, one of which is performed when the value of a condition expression is **true**, and the other when it is false. Let e be an expression and C and D be commands, then the alternative conditional form is defined as

$$[\![\texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ D]\!] = [\![\texttt{if}\ \{\ e\texttt{:}\ C\ \texttt{|}\ \texttt{\sim}e\texttt{:}\ D\ \}]\!]$$

5.3 Loop

$$\mathit{loop} = \texttt{while}\ \langle\text{expression}\rangle\ \texttt{do}\ \langle\text{command}\rangle$$

A *looping command* repetitively evaluates a condition expression and executes a command if the value of the expression is **true**. When it is **false**, it terminates.

6 Types, names and scope

6.1 Types

6.1.1 Primitive types

$$\begin{aligned} \textit{type} &= \langle \text{primitive-type} \rangle \\ \textit{primitive-type} &= \texttt{var} \end{aligned}$$

There is a single primitive *variable* type in sire. It holds a value that can be changed by input or assignment and it is specified by the keyword **`var`**. Variables are interpreted as signed integers and if no value has been assigned to it then the value is arbitrary.

6.1.2 Array types

$$\begin{aligned} \textit{type} &= \langle \text{array-type} \rangle \\ \textit{array-type} &= \langle \text{primitive-type} \rangle \\ &\mid \langle \text{array-type} \rangle \ \texttt{[} \ \langle \text{expression} \rangle \ \texttt{]} \end{aligned}$$

An *array variable* contains a number of component variables. Let T be a primitive type, then a 1-dimensional array of components of type T is specified by the type $T[e]$ where e is an expression specifying the length of the dimension. Arrays with additional dimensions can be declared by specifying additional lengths in the type. Let T be a primitive type, then $T[e_1][e_2]\cdots[e_d]$ specifies a d-dimensional array with $e_1 \times e_2 \times \cdots \times e_d$ components of type T.

The components of an array variable are accessed with integer-valued subscripts that can appear as an element in an expression or on the left hand side of an assignment or input. The *element* $u[i]$ selects the i^{th} component of a 1-dimensional array variable with the name u, and the subscript $v[i_1][i_2]\cdots[i_d]$ selects the i_1^{th} component of the first dimension and the i_2^{th} component of the second dimension etc. of a d-dimensional array variable with the name v.

6.2 Scope

A name has a context in which it is valid and can be used. This is called the *scope*. The scope of a name extends from the point at which it is declared to the end of the *block*. A block is enclosed by curly braces **{**$\cdots$**}** or a choice or alternative. The scope of a name also extends to any nested blocks.

6.3 Declarations

$$\textit{declaration} = \langle\text{type}\rangle\ \{_1\ \texttt{,}\ \langle\text{name}\rangle\ \}$$

A *declaration* $T\ n$ declares a variable of type T with the name n. A single declaration can specify multiple names of a particular type. Let $S(x)$ be a scope in which the name x is free, T be a type and $N_1, N_2, \cdots, N_n$ be names, then

$$[\![T\ \ N_1, N_2, \cdots, N_n\ \texttt{:}\ S]\!] = [\![T\ \ N_1\ \texttt{:}\ T\ \ N_2\ \texttt{:}\ \cdots\ \texttt{:}\ T\ \ N_n\ \texttt{:}\ S]\!]$$

6.4 Abbreviations

$$\begin{aligned}
\textit{abbreviation} &= \langle\text{specifier}\rangle\ \langle\text{name}\rangle\ \texttt{is}\ \langle\text{element}\rangle \\
&\mid \texttt{val}\ \langle\text{name}\rangle\ \texttt{is}\ \langle\text{expression}\rangle \\
\textit{specifier} &= \langle\text{primitive-type}\rangle \\
&\mid \langle\text{specifier}\rangle\ \texttt{[ ]} \\
&\mid \langle\text{specifier}\rangle\ \texttt{[}\ \langle\text{expression}\rangle\ \texttt{]} \\
\textit{element} &= \langle\text{element}\rangle\ \texttt{[}\ \langle\text{expression}\rangle\ \texttt{]} \\
&\mid \langle\text{field}\rangle \\
&\mid \langle\text{name}\rangle
\end{aligned}$$

An *abbreviation* can be used to specify new names for variables and values.

6.4.1 Variable abbreviation

A *variable abbreviation* specifies the name for an *element* that can either be a name or subscripted component of an array. Let T be a type, n be a name, e be an element and S be a scope. Then a variable abbreviation has the effect

$$[\![T\ \ n\ \texttt{is}\ e\ \texttt{:}\ S(n)]\!] = [\![S(e)]\!]$$

The length specifier for an array dimension can be omitted from an abbreviation to define a name of an array of components of the specified type of any length in that dimension. An array abbreviation is said to be *compatible* with an array variable if the lengths of all dimensions are equal or the abbreviation does not specify the length of a particular dimension.

6.4.2 Value abbreviation

A *value abbreviation* specifies a name for an expression. Let n be a name, e be an expression and S be a scope. Then a value abbreviation has the effect

$$[\![\texttt{val}\ \ n\ \texttt{is}\ e\ \texttt{:}\ S(n)]\!] = [\![S(e)]\!]$$

The value of the abbreviation must remain constant throughout its use. To ensure this, the abbreviated expression must not contain any variables that are updated by assignment or input in the scope of the abbreviation.

6.5 Specifications

$$
\begin{aligned}
\textit{specification} &= \langle \text{declaration} \rangle \\
&\mid \langle \text{abbreviation} \rangle \\
\textit{command} &= \langle \text{specification} \rangle \ \mathbf{:} \ \langle \text{command} \rangle \\
\textit{alternative} &= \langle \text{specification} \rangle \ \mathbf{:} \ \langle \text{alternative} \rangle
\end{aligned}
$$

A *specification* introduces a new name into a scope and all names specified within a block must be unique, however, a block can introduce a name that is already in scope. In general, a free variable of a block is *bound* to the most recent name that is still in scope.

If $S(x)$ and $S(y)$ are scopes with the same behaviour, but $S(x)$ contains the free variable x wherever $S(y)$ contains the free variable y and vice versa, and $N(x)$ and $N(y)$ are identical specifications except that $N(x)$ specifies the name x and $N(y)$ specifies the name y, then

$$[\![N(x) \ \mathbf{:} \ S(x)]\!] \ = \ [\![N(y) \ \mathbf{:} \ S(y)]\!]$$

In other words, specifications can always be added to produce a block in which the free variables are distinct from any scope it may be inserted into. This provides a basis for the semantics of substitution for process definitions, server definitions and functions, which are all explained in §11, p. 22.

6.6 Rules

1. *Declarations.* A specification can only introduce declarations that have a primitive variable type.
2. *Valid abbreviations.* The specifier of the element specified in the abbreviation must be compatible with the type of the name. Compatibility requires the primitive types to be the same and, for array types, each specified length must be also be equal in the corresponding dimension (unspecified lengths can match any dimension).
3. *Abbreviation subscripts.* Any variables used in subscripts of an abbreviated variable cannot be changed in the scope of the abbreviation.
4. *Value abbreviations.* The value of an abbreviated variable may be changed but to ensure the abbreviation always refers to the same array component, then no variable in the subscript expression of that component can be changed by assignment or input.
5. *Array abbreviations.* Components of an array must be identified by a single name within a given scope to prevent aliasing. Therefore, components of an array that has one or more abbreviated components may also be referred to by abbreviations.

7 Composition

$$\begin{aligned} \textit{command} &= \texttt{\{}\ \langle\text{sequence}\rangle\ \texttt{\}} \\ &\mid \texttt{\{}\ \langle\text{parallel}\rangle\ \texttt{\}} \end{aligned}$$

7.1 Sequence

$$\textit{sequence} = \{_0\ \texttt{;}\ \langle\text{command}\rangle\ \}$$

A set of commands are composed in *sequence* with the '**;**' separator. Execution starts with the first component command and each subsequent command is executed if and when the preceding command terminates. The sequence terminates when the last component command terminates.

7.2 Parallel

$$\begin{aligned} \textit{parallel} &= \{_0\ \texttt{\&}\ \langle\text{parallel-component}\rangle\ \} \\ \textit{parallel-component} &= \langle\text{process}\rangle \end{aligned}$$

A *parallel command* creates new *processes* that execute in parallel, where a process is another command. A set of processes are composed in parallel with the '**&**' separator. It causes the component processes to be executed simultaneously and it terminates if and only if all of the component processes have terminated.

7.2.1 Process interfaces

$$\begin{aligned} \textit{process} &= \langle\text{interface}\rangle\ \texttt{:}\ \langle\text{command}\rangle \\ &\mid \langle\text{command}\rangle \\ \textit{interface} &= \texttt{interface (}\ \{_0\ \texttt{,}\ \langle\text{declaration}\rangle\ \}\ \texttt{)} \\ \textit{type} &= \langle\text{chanend-type}\rangle \\ \textit{chanend-type} &= \texttt{chanend} \end{aligned}$$

Processes executing in parallel can communicate via channels and a process can specify a number of local channel ends to do this. The set of *local* channel ends belonging to a process constitutes its *interface*, which is specified before the body of a process.

Let $N_1, N_2, \cdots, N_n$ be names, then

interface (chanend N_1**,** N_2**,** $\cdots$**,** N_n**)**

specifies an interface that declares n channel ends in the scope of the process. No other types may be declared in a process interface.

7.2.2 Process names

$$\begin{aligned} \textit{parallel-component} &= \langle\text{process-label}\rangle\ \langle\text{process}\rangle \\ \textit{process-label} &= \langle\text{name}\rangle\ \texttt{is} \end{aligned}$$

Channel connections are established between component processes of a parallel command by *naming* the processes. This is done by prefixing a process with a *label* of the form '⟨name⟩ **is**'. The name of a process acts as a prefix for a *compound name* for the channels in the interface, where each one can be selected as a *field*. Let P be a process and $N_1, N_2, \cdots, N_n$ be names, then

$$\texttt{p is interface (chanend } N_1\texttt{, } N_2\texttt{, } \cdots\texttt{, } N_n\texttt{): } P$$

is a process with the name `p` that specifies an interface with n components. Each component of the interface can be selected with the fields $\texttt{p}.N_1, \texttt{p}.N_2, \cdots, \texttt{p}.N_n$.

A process name is visible to all of the component processes in the parallel command and a process that is not named and does not engage in any channel communication is said to be *anonymous*. A process can make a connection to another parallel process by selecting the channel end name as a field using the process' name and using it as the target of a connect statement.

7.2.3 Process arrays

Arrays of processes can be constructed provided they have the same interface. Components of a process array are selected using integer-valued subscripts. Let $P_1, P_2, \ldots, P_n$ be processes with the same interface, then

$$\texttt{p is \{}P_1\texttt{, } P_2\texttt{, } \cdots\texttt{, } P_n\texttt{\}}$$

declares a *process array* containing n processes. The component processes are selected with the subscripts `p[0]` to $\texttt{p[}n-1\texttt{]}$. Process arrays can also be nested to create multidimensional arrays.

7.3 Rules

6. *Process disjointness.* Processes in a parallel command must be *disjoint* and only update disjoint sets of variables. This is to eliminate the possibility of unwanted race conditions, i.e. in the absence of alternation (which is explained in the next section), leaving alternation as the only means of introducing non-determinism.

   A variable or component of an array variable can be *changed* by assignment or input, or *read* if it occurs as an operand or in an expression. Disjointness is enforced by permitting *read-only access* to variables or components of array variables shared between components of a parallel command, and *exclusive write access* when a variable or array variable component appears in at most one component of a parallel command. With

arrays, components of an array variable may only be changed in parallel if it can be determined at compile time that the array subscripts are used in a linear combination to select disjoint components of the array.

7. *Point-to-point channels.* No channel end can be connected to by more than one process. This is to ensure that no channel can be used for input or output by more than one process.

8. *Channel end subscripts.* If the target of a connect command is a component of a process array, the name will contain a subscript and the subscript must only use constants or replicator indices. This is so that the process to which it belongs can be identified at compile time.

9. *Nesting.* Channel connections can only be established between component processes of a parallel. Therefore, channel end names can only be used by child processes of the single parallel command and they cannot be passed to parallel commands nested in a process.

10. *Named process arrays.* Components of a named array must all present the same interface.

11. *Process interfaces.* A process interface can only declare channel end components.

8 Servers

A *server* is a special type of process that consists of special declarations that define its behaviour. It executes in parallel with the processes in its scope, is only active in response to *calls* and terminates when control flow leaves the block in which it is defined. *Client* processes of a server make calls that behave in the same way to a local procedure call.

Servers provide a mechanism for abstraction by building program components that provide a service or *subroutine*, and they can be combined into larger *server structures* to employ parallelism or distribute storage.

8.1 Specification

$$\begin{aligned} \textit{server} &= \langle\text{interface}\rangle \texttt{ : } \langle\text{server-specification}\rangle \\ \textit{server-specification} &= \langle\text{declaration}\rangle \\ &\quad \mid \texttt{ \{ } \{_1 \texttt{ : } \langle\text{declaration}\rangle \texttt{ \} } \texttt{ \}} \end{aligned}$$

The specification of a server consists of an interface that defines means of external interaction (consisting of calls and channel ends) and set of declarations that define its behaviour. These are *initialisation commands* that are performed before it responds to anything, responses to each communication or call and *finalisation commands* that are performed when it terminates.

8.1.1 Interface

$$\begin{aligned} \textit{declaration} &= \texttt{call } \{_1 \texttt{ , } \langle\text{name}\rangle \texttt{ (} \{_0 \texttt{ , } \langle\text{formal}\rangle \texttt{ \}) \}} \\ \textit{abbreviation} &= \texttt{call } \langle\text{name}\rangle \texttt{ is } \langle\text{name}\rangle \end{aligned}$$

A call declaration

$$\texttt{call } N \texttt{ (}f_1, f_2, \cdots, f_n\texttt{)}$$

specifies N as a name for a call with the formal parameters $f_1, f_2, \cdots, f_n$. A single call declaration can specify multiple calls. Let F_k be a sequence of formals $f_1, f_2, \cdots, f_n$ where n is an integer defined for the sequence.

$$\begin{aligned} &[\![\texttt{call } N_1 \texttt{ (}F_1\texttt{), } N_2\texttt{, (}F_2\texttt{), } \cdots\texttt{, } N_n \texttt{ (}F_n\texttt{)}]\!] = \\ &\quad [\![\texttt{call } N_1 \texttt{ (}F_1\texttt{), call } N_2\texttt{, (}F_2\texttt{), } \cdots\texttt{, call } N_n \texttt{ (}F_n\texttt{)}]\!] \end{aligned}$$

8.1.2 Alternation and call guards

$$
\begin{aligned}
\textit{declaration} &= \langle \text{alternation} \rangle \\
\textit{guard} &= \texttt{accept}\ \langle \text{name} \rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle \text{formal} \rangle\ \}\ \texttt{)} \\
&\mid \langle \text{expression} \rangle\ \texttt{\&}\ \texttt{accept}\ \langle \text{name} \rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle \text{formal} \rangle\ \}\ \texttt{)}
\end{aligned}
$$

The main behaviour of a server is defined by an *alternation declaration* that specifies an alternation command (see §5.1, p. 7 for the definition of alternation). This, as well as containing input alternatives, can contain *call* alternatives.

An *alternative* dealing with a call specifies the formal parameters that must match the list of formals in the specification in the interface of the corresponding call exactly, including the names, and a command to be executed when the call is invoked by a client. A call alternative may be guarded with an expression in the same way as an input can, so that the call is ready only when the guard is **`true`**. The effect is that the server will not respond to a call while the guard is false.

8.1.3 Initialisation and finalisation

$$
\begin{aligned}
\textit{declaration} &= \texttt{initial}\ \langle \text{command} \rangle \\
&\mid \texttt{final}\ \langle \text{command} \rangle
\end{aligned}
$$

An *initialisation declaration* specifies a command to be executed before the alternation. A *finalisation declaration* specifies a command to be executed after the alternation, when the scope of the server has terminated.

8.2 Declarations

$$
\begin{aligned}
\textit{declaration} &= \langle \text{server-declaration} \rangle \\
&\mid \langle \text{hiding-declaration} \rangle \\
&\mid \langle \text{simultaneous-declaration} \rangle
\end{aligned}
$$

8.2.1 Server declarations

$$
\begin{aligned}
\textit{server-declaration} &= \langle \text{name} \rangle\ \texttt{is}\ \langle \text{server} \rangle \\
\textit{server} &= \langle \text{server-array} \rangle \\
\textit{server-array} &= \texttt{[}\ \{_1\ \texttt{,}\ \langle \text{server} \rangle\ \}\ \texttt{]}
\end{aligned}
$$

A *server declaration*

`n is` S

declares **n** as a name for the server S and **n** acts as a prefix for the compound names of the call and channel names specified in the interface of the server.

Arrays of servers can be constructed provided they have the same interface. Let $S_1, S_2, \ldots, S_n$ be servers with the same interface, then

s is [S_1, S_2, $\cdots$, S_n]

declares a server array containing n servers that are selected with the subscripts **s[0]** to **s**[$n-1$]. Server arrays can also be nested to create multidimensional arrays.

8.2.2 Hiding declarations

hiding-declaration = **from {** {$_1$ **:** ⟨declaration⟩ } **} interface** ⟨name⟩

A *hiding declaration* encloses a set of declarations and specifies one name declared by these to be visible to the scope of the declaration. This is used to create minimal interfaces with collections of servers to perform abstraction.

8.2.3 Simultaneous declarations

simultaneous-declaration = {$_0$ **&** ⟨declaration⟩ }

Declarations separated by the symbol '**&**' occur *simultaneously* and have the same scope. They are therefore visible to one another. Simultaneous declarations are used to introduce mutually referential declarations, such as servers that communicate with each other.

8.3 Calls

command = ⟨element⟩ **(** {$_0$**,** ⟨actual⟩ } **)**

A *server call* is a command that specifies the name of the server, name of the call and a list of actual parameters. The behaviour of a server call is defined in the following way. Let $f_1, f_2, \ldots, f_n$ be formals, $a_1, a_2, \ldots, a_n$ be actuals, and C be a command; then

s is interface (call c(f_1, f_2, $\cdots$, f_n)**):**
alt { **accept** c(f_1, f_2, $\cdots$, f_n) C **}:**
s.c (a_1, a_2, $\cdots$, a_n)

has the effect

f_1 **is** a_1 **:** f_2 **is** a_2 **:** $\ldots$ **:** f_n **is** a_n **:** C

8.4 Rules

12. *Server channels.* A server can use channels for communication and these are subject to the same rules as processes, except that connections can only be established with other servers with the same scope. Since channel inputs can only appear in the alternatives in an alternation declaration, output commands can only appear either in the command associated with an alternative or in the initialisation or finalisation commands.

13. *Server calls.* A server must provide call guards for each of the calls specified in its interface.

14. *Server interfaces.* A server interface can only contain channel end and call specifier components.

15. *Server disjointness.* Servers are subject to the same variable- and channel-disjointness rules as processes (see §7.3, p. 13). In particular, a server can change a variable or component of an array by assignment or input if no other server or process in its scope can read or change that variable, and, a server can read the value from a variable or component of an array only if no other server or process changes the variable.

16. *Named server arrays.* Components of a server array must all present the same interface.

9 Replication

$$
\begin{aligned}
\textit{command} &= \textbf{seq}\ \langle\text{replicator}\rangle\ \langle\text{command}\rangle \\
\textit{process} &= \textbf{par}\ \langle\text{replicator}\rangle\ \langle\text{process}\rangle \\
\textit{conditional} &= \textbf{if}\ \langle\text{replicator}\rangle\ \langle\text{choice}\rangle \\
\textit{alternation} &= \textbf{alt}\ \langle\text{replicator}\rangle\ \langle\text{alternative}\rangle \\
\textit{server-declaration} &= \langle\text{name}\rangle\ \textbf{is}\ \langle\text{replicator}\rangle\ \langle\text{server}\rangle \\
&\mid \langle\text{name}\rangle\ \textbf{is}\ \textbf{[}\ \langle\text{expression}\rangle\ \textbf{]}\ \langle\text{server}\rangle \\
\textit{replicator} &= \textbf{[}\ \{_1\ \textbf{,}\ \langle\text{index-range}\rangle\ \}\ \textbf{]} \\
\textit{index-range} &= \langle\text{name}\rangle\ \textbf{=}\ \langle\text{expression}\rangle\ \textbf{for}\ \langle\text{expression}\rangle \\
&\mid \langle\text{name}\rangle\ \textbf{=}\ \langle\text{expression}\rangle\ \textbf{for}\ \langle\text{expression}\rangle\ \textbf{step}\ \langle\text{expression}\rangle
\end{aligned}
$$

Replicators can be used to create a number of similar commands, components of a construct, processes or servers. A replicator consists of one or more *index ranges*. An index range declares the name of an *index* variable, a *base* expression, a *count* expression and, optionally, a *step* expression. The base, count and step define the range of values that an index variable takes. This name of the index variable is available to the replicated component and each replicated instance takes a unique value of the index.

9.1 Constructs

Let X be one of **if**, **alt**, **seq** or **par**; $Y(i)$ be a choice, alternative, command or process corresponding to X in which i is free; $\oplus$ be one of the separators '|', '**;**' or '**&**' corresponding to X; and b, c and s be integer values. Then, the behaviour of a *replicated conditional, alternative, sequence or parallel* with a single index range is defined by the following. If $c \geq 0$:

$$[\![X\ \texttt{[}i\texttt{=}b\ \textbf{for}\ c\ \textbf{step}\ s\texttt{]}\ Y(i)]\!] = [\![X\ \texttt{\{}\ Y(b) \oplus Y(b+s) \oplus \cdots \oplus Y(b+(c-1)s)\ \texttt{\}}]\!]$$

If $c = 0$:

$$[\![\textbf{if}\ \texttt{[}i\texttt{=}b\ \textbf{for}\ \texttt{0}\ \textbf{step}\ s\texttt{]}\ Y(i)]\!] = [\![\textbf{stop}]\!]$$

$$[\![\textbf{alt}\ \texttt{[}i\texttt{=}b\ \textbf{for}\ \texttt{0}\ \textbf{step}\ s\texttt{]}\ Y(i)]\!] = [\![\textbf{stop}]\!]$$

$$[\![\textbf{seq}\ \texttt{[}i\texttt{=}b\ \textbf{for}\ \texttt{0}\ \textbf{step}\ s\texttt{]}\ Y(i)]\!] = [\![\textbf{skip}]\!]$$

$$[\![\textbf{par}\ \texttt{[}i\texttt{=}b\ \textbf{for}\ \texttt{0}\ \textbf{step}\ s\texttt{]}\ Y(i)]\!] = [\![\textbf{skip}]\!]$$

If $c < 0$:

$$[\![X\ \texttt{[}i\texttt{=}b\ \textbf{for}\ c\ \textbf{step}\ s\texttt{]}\ Y(i)]\!] = [\![\textbf{stop}]\!]$$

For an index range without a step expression

$$[\![X\ \texttt{[}i\texttt{=}b\ \textbf{for}\ c\texttt{]}\ Y(i)]\!] = [\![X\ \texttt{[}i\texttt{=}b\ \textbf{for}\ c\ \textbf{step}\ \texttt{1]}\ Y(i)]\!]$$

Let I_k be a index range $i_k = b_k$ **for** c_k, where i_k is a name and b_k and c_k are integer values defined for the index range. Let b_i and c_i for $1 \leq i \leq d$ be integer values, then the behaviour of *replicators* with d index ranges is defined by

$$[\![X \texttt{ [} I_1, I_2, \cdots, I_d \texttt{]} \ Y(i_1, i_2, \cdots, i_d)]\!] = \\ [\![X \texttt{ [} I_1 \texttt{]} \texttt{ \{ } X \texttt{ [} I_2 \texttt{]} \texttt{ \{ } \cdots \texttt{ \{ } X \texttt{ [} I_d \texttt{]} \ Y(i_1, i_2, \cdots, i_d) \texttt{ \}\}\}}]\!]$$

9.2 Servers

Servers can also be replicated. Let $S(i)$ be a server specification in which i is free; n be a name; and b, c and s be integer values. Then, the behaviour of a *replicated server* with a single index range is defined by

$$[\![\texttt{[}i = b \textbf{ for } c \textbf{ step } s\texttt{]} \ S(i)]\!] = [\![\texttt{[}S(b)\texttt{, } S(b+s)\texttt{, } \cdots\texttt{, } S(b+(c-1)s)\texttt{]}]\!]$$

$$[\![\texttt{[}i = b \textbf{ for } c\texttt{]} \ S(i)]\!] = [\![\texttt{[}i = b \textbf{ for } c \textbf{ step } \texttt{1]} \ S(i)]\!]$$

Let I_k be a index range $i_k = b_k$ **for** c_k, where i_k is a name and b_k and c_k are integer values. Then, the behaviour of replicators with d index ranges is defined by

$$[\![\texttt{[}I_1\texttt{, } I_2\texttt{, } \cdots\texttt{, } I_d\texttt{]} \ N]\!] = [\![\textbf{n is } \texttt{[}I_1\texttt{]} \texttt{ [[}I_2\texttt{]} \texttt{ [} \cdots \texttt{ [[}I_d\texttt{]} \ N \texttt{]]]}]\!]$$

When replicated instances of a server do not require their unique indices, then a shorthand form of the replicator can be used. Let e be an expression, then

$$[\![\texttt{[}e\texttt{]} \ N]\!] = [\![\texttt{[}i = 0 \textbf{ for } e\texttt{]} \ N]\!]$$

9.3 Rules

17. *Replicator indices.* The value of any replicator index cannot be changed by an assignment or input.

10 Expressions and elements

An *expression* produces a value and is composed of operands and operators. They cannot cause any *side-effects* by changing any external state. An *operand* is either an element or literal or nested expression, which is enclosed by parentheses. There are no precedence rules, so nested expressions must be explicitly bracketed. An *operator* takes either one or two operands and produces a value. An *element* is either a name, subscripted name, field, or function call.

A full specification of expressions and elements is given in §??, p. ??.

10.1 Valof expressions

$$\begin{aligned} \textit{valof} &= \texttt{valof}\ \langle\text{command}\rangle\ \texttt{result}\ \langle\text{expression}\rangle \\ &\mid\ \langle\text{specification}\rangle\ \texttt{:}\ \langle\text{valof}\rangle \\ \textit{expression} &= \texttt{(}\ \langle\text{valof}\rangle\ \texttt{)} \end{aligned}$$

A *valof* expression produces a value from a command. A valof expression **valof** C **result** e is a block and the scope of C extends to the **result** expression e. It is evaluated by performing the command C and then evaluating the expression e to produce the result.

10.2 Rules

18. *Valof side-effects.* A valof expression cannot cause side-effects by changing any external state. It cannot therefore make calls using a free name or assign to a free variable.

11 Procedural abstraction

The details of a process, server or expression can be hidden in a component that presents an interface to allow the behaviour to be considered abstractly. These components are called *procedures* and *functions*.

11.1 Procedures

11.1.1 Definition

$$
\begin{aligned}
\textit{definition} &= \langle \text{procedure} \rangle \\
\textit{procedure} &= \texttt{process}\ \langle \text{name} \rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle \text{formal} \rangle\ \}\ \texttt{)}\ \texttt{is}\ \langle \text{process} \rangle \\
&\mid \texttt{server}\ \langle \text{name} \rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle \text{formal} \rangle\ \}\ \texttt{)}\ \texttt{is}\ \langle \text{server} \rangle \\
\textit{formal} &= \langle \text{specifier} \rangle\ \{_1\ \texttt{,}\ \langle \text{name} \rangle\ \} \\
&\mid \texttt{val}\ \{_1\ \texttt{,}\ \langle \text{name} \rangle\ \}
\end{aligned}
$$

A procedure specifies a name, a process or server and a set of *formal parameters*. Since there is no global scope in sire the set of formal parameters are the free elements of the process or server.

Let S be a specifier, then a formal parameter of the form $S\ n$ specifies a name n with type given by S. A single formal can also specify multiple names. Let S be a specifier and $N_1, N_2, \cdots, N_n$ be names, then

$$S\ \ N_1, N_2, \cdots, N_n\ =\ S\ \ N_1\texttt{,}\ \ S\ \ N_2\texttt{,}\ \ \cdots\texttt{,}\ \ S\ \ N_n$$

S may specify a variable or array variable of a primitive or channel end type, or a compound channel end or array of compound channel ends. For value specifications

$$\texttt{val}\ \ N_1, N_2, \cdots, N_n\ =\ \texttt{val}\ \ N_1\texttt{,}\ \ \texttt{val}\ \ N_2\texttt{,}\ \ \cdots\texttt{,}\ \ \texttt{val}\ \ N_n\texttt{.}$$

Let X be one of **process** or **server**, Y be a process or server corresponding to X and $f_1, f_2, \cdots, f_n$ be formal parameters. Then, the definition

$$X\ \ N\ \ \texttt{(}f_1, f_2, \cdots, f_n\texttt{)}\ \ \texttt{is}\ \ Y$$

defines the name N to be a procedure that behaves like Y. If f_i is an array type, it must specify the length of each dimension with an expression that consists of only of constant values or variables specified by other the other formals $f_1, f_2, \cdots, f_{i-1}, f_{i+1}, \cdots, f_n$.

11.1.2 Channel end and server call parameters for procedures

$$
\begin{aligned}
\textit{primitive-type} &= \langle\text{process-type}\rangle \\
&\mid \langle\text{server-type}\rangle \\
\textit{process-type} &= \textbf{process}\ \langle\text{name}\rangle \\
&\mid \textbf{process}\ \langle\text{interface}\rangle \\
\textit{server-type} &= \textbf{server}\ \langle\text{name}\rangle \\
&\mid \textbf{server}\ \langle\text{interface}\rangle
\end{aligned}
$$

A procedure can specify parameters for channel ends and server calls. These may be specified directly or with a compound name from a process or server reference with the keywords **process** or **server** respectively, its interface and a name. The interface can either be specified *explicitly* by listing the components, or *implicitly* by giving the name of a procedure that defines a process or server with the interface.

11.1.3 Passing procedures as parameters

$$
\begin{aligned}
\textit{formal} &= \langle\text{call-type}\rangle\ \{_1\ \texttt{,}\ \langle\text{name}\rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{formal}\rangle\ \}\ \texttt{)}\ \} \\
\textit{abbreviation} &= \langle\text{call-type}\rangle\ \langle\text{name}\rangle\ \texttt{(}\ \{_0\ \texttt{,}\ \langle\text{formal}\rangle\ \}\ \texttt{)}\ \textbf{is}\ \langle\text{name}\rangle \\
\textit{call-type} &= \textbf{process} \\
&\quad\ \textbf{function}
\end{aligned}
$$

A procedure or server call can specify procedures as parameters. This is so that different implementations of a procedure with the same formals can be supplied to a procedure instance or server call. With a procedure parameter to a server call, the supplied procedure is executed remotely.

A formal parameter of the form

$$\textbf{process}\ N\ \texttt{(}f_1\texttt{,}\ f_2\texttt{,}\ \cdots\texttt{,}\ f_n\texttt{)}$$

specifies N as the name for a procedure parameter. A single procedure can specify multiple calls. Let F_k be a sequence of formals $f_1, f_2, \cdots, f_n$ where n is an integer defined for the sequence.

$$
\begin{aligned}
&[\![\textbf{process}\ N_1\ \texttt{(}F_1\texttt{)}\texttt{,}\ N_2\texttt{,}\ \texttt{(}F_2\texttt{)}\texttt{,}\ \cdots\texttt{,}\ N_n\ \texttt{(}F_n\texttt{)}]\!] = \\
&\quad [\![\textbf{process}\ N_1\ \texttt{(}F_1\texttt{)}\texttt{,}\ \textbf{process}\ N_2\texttt{,}\ \texttt{(}F_2\texttt{)}\texttt{,}\ \cdots\texttt{,}\ \textbf{process}\ N_n\ \texttt{(}F_n\texttt{)}]\!]
\end{aligned}
$$

11.2 Instances

$$
\begin{aligned}
\textit{process} &= \langle\text{instance}\rangle \\
\textit{server} &= \langle\text{instance}\rangle \\
\textit{command} &= \langle\text{instance}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\textit{instance} &= \langle \text{name} \rangle \texttt{ (} \{_0 \texttt{ , } \langle \text{actual} \rangle \} \texttt{)} \\
\textit{actual} &= \langle \text{element} \rangle \\
&\mid \langle \text{expression} \rangle
\end{aligned}
$$

An *instance* of a procedure is created by specifying its name and a list of *actual parameters*. When an *instance* of the procedure is created, each formal serves as the left hand side of an abbreviation of an *actual parameter*, a, that is supplied to the instance. This provides a binding of each free variable to the scope in which it is instantiated. Each actual is an element or expression with a compatible type with the formal parameter.

The scoping rules mean that an instance of a process type can be substituted in-place for the process it names by inserting abbreviations of each formal with the actual parameter. This is defined in the following way. Let X be one of **process** or **server**, Y be a process or server corresponding to X, $f_1, f_2, \cdots, f_n$ be formals, $a_1, a_2, \cdots, a_n$ be actuals. Then, if X is a program in which no name is specified more than once and it contains the definition

$$X \;\; N \;\; \texttt{(}f_1, f_2, \cdots, f_n\texttt{)} \;\; \textbf{is} \;\; Y$$

then, within its scope

$$N \;\; \texttt{(}a_1, a_2, \ldots, a_n\texttt{)} \;=\; f_1 \;\; \textbf{is} \;\; a_1 \;\; \texttt{:} \;\; f_2 \;\; \textbf{is} \;\; a_2 \;\; \texttt{:} \;\; \ldots \;\; \texttt{:} \;\; f_n \;\; \textbf{is} \;\; a_n \;\; \texttt{:} \;\; P$$

provided each abbreviation is valid. This allows procedures to be compiled either as a closed subroutine or by substituting the body of the definition directly with the instance.

11.3 Hiding definitions

$$\textit{definition} \;=\; \textbf{server} \;\langle \text{name} \rangle \;\texttt{(}\; \{_0 \texttt{,} \;\langle \text{formal} \rangle\; \}\; \texttt{)} \;\textbf{inherits}\; \langle \text{hiding-declaration} \rangle$$

A different way to define a server is to inherit an interface from a server in a set of declarations. This can be used to combine a number of servers into a single reusable module, exposing a single server declaration as an interface.

Let $i_1, i_2, \cdots, i_m$ be specifiers for components of an interface, then the definition

server N**(**f_1**,** f_2**,** $\cdots$**,** f_n**) inherits**
from
⋮
s **is interface** (**(**i_1**,** i_2**,** $\cdots$**,** i_m**)** **to** D
⋮
interface s

defines N as the name of a server type who inherits the interface from the server with name **s** in the hiding declaration. Let X be a program in form where no name is specified more than once, then if X contains the above definition, in the scope of N

$$[\![\texttt{x is } N(a_1, a_2, \cdots, a_n) \texttt{ : } S]\!] =$$
$$[\![\ f_1 \texttt{ is } a_1 \texttt{ : } f_2 \texttt{ is } a_2 \texttt{ : } \ldots \texttt{ : } f_n \texttt{ is } a_n \texttt{ :}$$
$$\texttt{ s is interface}(i_1, i_2, \cdots, i_n)\texttt{: } D \texttt{ : } \cdots \texttt{ : } S]\!]$$

where S is the scope of the declaration.

11.4 Functions

11.4.1 Definitions and instances

$$\textit{definition} = \textbf{function}\ \langle\text{name}\rangle\ \texttt{(}\ \{_0\texttt{,}\ \langle\text{formal}\rangle\ \}\ \texttt{)}\ \textbf{is}\ \langle\text{valof}\rangle$$
$$\textit{expression} = \langle\text{instance}\rangle$$

A valof expression can be reused by defining a *function*. Each formal parameter of a function must be a value abbreviation and functions cannot contain calls to processes. The definition

$$\textbf{function}\ N\ \ (f_1, f_2, \cdots, f_n)\ \textbf{is valof}\ C\ \textbf{result}\ e$$

defines N as the name of a function with the valof expression **valof** C **result** e.

Let X be a program in a form where no name is specified more than once, then if X contains the above function definition, in the scope of N

$$[\![N\ \ (a_0, a_1, \cdots, a_n)]\!] =$$
$$[\![\texttt{(}\textbf{valof}\ f_0\ \textbf{is}\ a_0\ \texttt{:}\ f_1\ \textbf{is}\ a_1\ \texttt{:}\ \cdots\ \texttt{:}\ f_n\ \textbf{is}\ a_n\ \texttt{:}\ C\ \textbf{result}\ e\texttt{)}]\!]$$

provided each abbreviation is valid. This allows functions to be compiled either as a closed subroutine or by substitution.

11.4.2 Functions as parameters

$$\textit{call-type} = \textbf{function}$$

Functions can be passed as parameters in the same way as procedures.

11.5 Rules

19. *Parameters.* The rules for procedures, hiding definitions and functions follow those for abbreviations (see §6.6, p. 11). The following two rules further define compatibility for interface, call and procedure types.

20. *Matching interface parameters.* A compound name that references a single process or server can only be supplied as an actual parameter if it is compatible. This requires the interface to be the same as the one specified by the formal parameter.

21. *Matching call and procedure parameters.* A call or procedure name can only be supplied as an actual parameter if it is compatible. This requires (1) the type of the formal parameter are the same and (2) each of the types of the formals in the definition of the call or procedure are the same as the formals specified in the parameter (which are specified in braces).

22. *Array parameters.* Each array parameter of a process or server type must specify its length by an expression that can contain only constant values or the names of other value parameters.

23. *Recursion.* Recursive procedures or functions are not permitted.

12 Program

$$
\begin{aligned}
\textit{program} &= \langle \text{program-specification} \rangle \ \textbf{:} \ \langle \text{program} \rangle \\
&\mid \langle \text{sequence} \rangle \\
\textit{program-specification} &= \langle \text{specification} \rangle \\
&\mid \langle \text{definition} \rangle \\
\textit{definition} &= \langle \text{simultaneous-definition} \rangle \\
\textit{simultaneous-definition} &= \{_0 \ \textbf{\&} \ \langle \text{definition} \rangle \ \}
\end{aligned}
$$

A *program* is a single command sequence, or process, with a specification that can contain definitions of process types, server types and functions.

12.0.1 Simultaneous definitions

Definitions separated by the symbol **&** occur *simultaneously* and share the same scope; they are therefore visible to one another. Simultaneous definitions are used to introduce mutually referential definitions, such as processes or servers that communicate with each other.

12.1 Rules

24. *No global scope.* A program specification can only contain definitions and abbreviations.