OPUS 1.0

Reference Manual

Barbara Chapman[1] **Matthew Haines**[2] **Piyush Mehrotra**[3]
Erwin Laure[1] **Hans Zima**[1] **John Van Rosendale**[3]

[1]Institute for Software Technology and Parallel Systems
University of Vienna
Liechtensteinstrasse 22, A-1090, Vienna, Austria
{barbara,erwin,zima}@par.univie.ac.at

[2]Computer Science Department
University of Wyoming
Laramie, WY 82071-3682
haines@cs.uwyo.edu

[3]Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Mail Stop 132C
Hampton, VA 23681-0001
{pm,jvr}@icase.edu

October 17, 1997

Contents

1 Overview **1**

1.1 Goals of OPUS . . . 1
1.2 Fortran 90/HPF Binding . . . 1
1.3 New Features in OPUS . . . 2
1.4 Syntax Metalanguage . . . 3

2 Basic Language Specification **4**

2.1 Introduction . . . 4
2.2 SDA Types . . . 4
2.2.1 Syntax . . . 4
2.2.2 Introduction . . . 4
2.2.3 SDA Type Arguments and Specification Part . . . 4
2.2.4 SDA Methods Part . . . 5
2.3 SDA Declaration . . . 6
2.3.1 Syntax . . . 6
2.3.2 Semantics . . . 7

3 Method Activation **8**

3.1 Syntax . . . 8
3.2 Introduction . . . 8
3.3 Synchronous Method Activation . . . 8
3.4 Asynchronous Method Activation . . . 9
3.5 Intrinsic Methods . . . 11
3.5.1 Syntax . . . 11
3.5.2 Introduction . . . 12
3.5.3 Creating an SDA Object . . . 12
3.5.4 Initialization of an SDA Object . . . 14
3.5.5 Termination of an SDA Object . . . 14
3.5.6 Saving an SDA Object . . . 15
3.5.7 Access to an SDA's Data . . . 15

4 Synchronization **16**

4.1 SDA Synchronization . . . 16
4.2 Task Synchronization . . . 16
4.2.1 Syntax . . . 16
4.2.2 Semantics . . . 16

Appendix **19**

A Examples **19**

A.1 Barrier Synchronization . . . 19
A.2 Exchange of Signals Between Tasks . . . 20
A.3 Consumer/Producer Problem . . . 20
A.4 Readers/Writers Problem . . . 21

B Syntax **25**

B.1 SDA Types . . . 25
B.2 SDA Declaration . . . 26

B.3 Method Activation 26
B.4 Intrinsic Methods 27
B.5 Task Synchronization 28

1 Overview

This document specifies the form and establishes the interpretation of programs expressed in the OPUS coordination language. OPUS is designed as a set of extensions to HPF [2] and Fortran 90 [3].

1.1 Goals of OPUS

The goal of OPUS is to define language extensions for HPF supporting the coordination of data parallel tasks. OPUS introduces coarse grained *task parallelism* on top of fine grained data parallelism. Furthermore OPUS supports the coupling of multiple programs into complex multidisciplinary codes. To achieve these goals a set of requirements OPUS has to fulfill were identified:

- Separate programs should be "encapsulated" into modules in a way that respects their separate name spaces.
- Coupling between modules should be at the highest level (as opposed to having message-passing constructs throughout the code).
- Both task-level parallelism between modules, and data parallelism within each module should be expressible.
- Flexible and general synchronization mechanisms should be provided to allow the programmer maximal freedom in exploitation of task-level parallelism.

A profound discussion of multidisciplinary codes and the coupling of task and data parallelism can be found in [1].

Although OPUS is defined as a language extension to HPF its design enables the coupling of modules implemented in different languages by defining appropriate OPUS bindings. Moreover OPUS can be extended to provide support for heterogeneous systems. Future efforts will address these important issues.

1.2 Fortran 90/HPF Binding

The base language for OPUS is HPF [2]. The module concept of Fortran 90 [3] as well as the data parallel features of HPF make HPF a natural base for OPUS.

OPUS introduces a new language feature, the *ShareD Abstraction (SDA)* to HPF which enables the encapsulation of data parallel HPF tasks. ShareD Abstractions extend the module concept of Fortran 90. A set of intrinsic procedures (called *methods*) is introduced for manipulating SDAs. In order to ensure data consistency a new attribute **READ_ONLY** is provided for restricting data access. For synchronization purposes new language features are defined, namely the **WAIT**, **WAITANY** and the **TEST** statement.

Beside the coordination of data parallel HPF tasks OPUS may also be used in a purely task parallel manner by coordinating Fortran 90 tasks.

1.3 New Features in OPUS

OPUS extends HPF/Fortran 90 in several areas, including:

- Encapsulation features;
- Task parallel execution features;
- Task coordination features.

We give a short overview of these areas below.

Encapsulation Features The coordination of complex multidisciplinary codes, where separate modules might be written by different groups using different data structures and approaches, requires the possibility of encapsulating modules with a clean interface.

OPUS provides the *ShareD Abstraction (SDA)* construct, which encapsulates data and the methods manipulating the data. SDAs can be used as computation servers as well as shared data repositories. Although SDAs borrow heavily from the Fortran 90 "module" concept, they are conceptually data types and provide therefore more flexibility. Section 2 describes the encapsulation features.

Task Parallel Execution Features An efficient implementation of multidisciplinary codes requires different levels of parallelism: while the data parallelism available within a module can be efficiently exploited by state of the art data parallel languages such as HPF, this kind of applications also requires coarse grained task parallelism between modules.

The task parallel features of OPUS allow a parallel execution of SDA methods, which may internally make use of data parallelism, expressed through HPF directives. Thus OPUS provides a programming model which is able to exploit multiple levels of (nested) parallelism. Section 3 describes the task parallel execution features.

Task Coordination Features In order to make an efficient use of task parallel features the programmer must have the possibility of coordinating and synchronizing parallel tasks in a flexible and general way. OPUS provides both implicit and explicit synchronization features which give the programmer a high degree of flexibility. Section 4 describes the task coordination features.

1.4 Syntax Metalanguage

Notation The notation used in the syntax rules is a variation of Backus-Naur form and in accordance with the Fortran 90 standard document [3].

The syntactic metasymbols used are:

is	introduces a syntactic class definition
or	introduces a syntactic class alternative
[]	encloses an optional item
[]...	encloses an optionally repeated item which may occur zero or more times

Nonterminal symbols which are part of Fortran 90 are implicitly defined according to the Fortran 90 language specification [3] and annotated with the Fortran 90 rule number (prefix *R*). Nonterminal symbols which are part of High Performance Fortran [2] are accordingly annotated with the HPF rule number (prefix *H*).

Example:

sda-specification-part	**is**	*specification-part(R204)*
processor-request	**is**	`PROCESSORS=`*processors-decl-list(H330)*

Assumed syntax rules are:

xyz-list	**is**	*xyz* [, *xyz*] ...
xyz-name	**is**	*name(R304)*
integer-xyz	**is**	*xyz*
xyz-variable	**is**	*variable(R601)*

2 Basic Language Specification

2.1 Introduction

In this section we define the set of Fortran 90/HPF extensions that constitute the basic OPUS language. The heart of the language is the *ShareD Abstraction (SDA)* construct. Section 2.2 introduces the SDA type while Section 2.3 shows how to declare an SDA object.

2.2 SDA Types

2.2.1 Syntax

sda-type	**is**	*sda-stmt* [*sda-specification-part*] [*sda-method-part*] *end-sda-stmt*
sda-stmt	**is**	`SDA TYPE` *type-name* [`(`*dummy-arg-list(R1221)*`)`]
sda-specification-part	**is**	*specification-part(R204)*
sda-method-part	**is**	*contains-stmt(R1225) method* [*method*]...
method	**is**	*function-method*
	or	*subroutine-method*
function-method	**is**	*function-stmt(R1216)* [*condition-clause*] *subprogram-body* *end-function-stmt(R1218)*
condition-clause	**is**	`WHEN (`*scalar-logical-expr(R725)*`)`
subprogram-body	**is**	[*specification-part(R204)*] [*execution-part(R208)*] [*internal-subprogram-part(R210)*]
subroutine-method	**is**	*subroutine-stmt(R1220)* [*condition-clause*] *subprogram-body* *end-subroutine-stmt(R1222)*
end-sda-stmt	**is**	`END` [`SDA TYPE` [*type-name*]]

2.2.2 Introduction

The core of OPUS is the **ShareD Abstraction (SDA)** construct. An *SDA type* specifies an object structure, which contains data along with the **methods** (i.e. procedures) which manipulate this data. The syntax of the SDA type is modeled after the Fortran 90 module syntax and, in general, consists of the following components:

- a *name* and a list of *dummy arguments*, which can be used to parameterize the SDA type,
- a *specification part*, declaring the required types and data structures together with their distributions and other attributes, and
- a *method part* which specifies *methods* acting on the SDA's internal data structures.

The definition of an SDA represents a *parameterized SDA type.*

2.2.3 SDA Type Arguments and Specification Part

The dummy arguments of an SDA type specification can be used to parameterize the internal data structures of the SDA (including local arrays). They conform to Fortran 90 procedure conventions and must be of intent IN. When an SDA object is created (cf. Section 3.5), the dummy arguments are bound to actual arguments and a *fully specified* SDA type is obtained.

The specification part declares the internal data structure of an SDA. Its syntax and semantics is similar to the Fortran 90 module specification part. Note that SDA declarations can be also enclosed in the specification part, thus allowing data sharing between SDAs. All entities declared inside the SDA type are considered public unless explicitly specified as **PRIVATE**. In addition to the Fortran 90 syntax the attribute **READ_ONLY** can be used to restrict the access to a public variable. As it is obvious from the name, **READ_ONLY** entities may be read by any SDA having access to them, but such an entity may only be modified by the SDA in which it is declared.

Public variables of an SDA are visible and accessible to all tasks having access to the SDA. They can be accessed using intrinsic method calls (cf. Section 3.5) and every Fortran 90 read or write access is implicitly transformed to such a method call. Thus, in order to avoid incoherent states of the SDA's data, all operations manipulating an SDA have exclusive access to its data and are executed as atomic operations.

The specification part may also contain HPF-directives to enable a data parallel execution of the methods.

The example below shows the SDA type definition for a buffer using a fifo strategy, where the buffer is realized as an array. The dummy argument *size* is used to parameterize the size of the buffer (*fifo*). HPF directives may be included into the specification part to allow data parallel computation.

```
SDA TYPE buffer_type(size)
    !HPF$ PROCESSORS directives
    INTEGER :: size
    REAL :: fifo(0:size-1)
    INTEGER :: count=0
    INTEGER :: px=0, cx=0
    !HPF$ ALIGN & DISTRIBUTE directives
    ...
CONTAINS
    ! method part
END buffer_type
```

2.2.4 SDA Methods Part

Methods are arbitrary procedures (either functions or subroutines) and may have arguments of any intent, which are passed with copy-in/copy-out semantics.

A *condition clause* (or *when condition*) may be associated with a method, which contains a side-effect free logical expression that controls the execution of the method in the sense that a call to the method can be executed only if the method expression evaluates to **true**. If it is **false** the method activation request is enqueued until the condition evaluates to true.

Method Arguments

The dummy arguments of a method fall into three categories:

- Arguments conforming to Fortran 90 conventions
  As mentioned above, these arguments may have an arbitrary intent and are passed with copy-in/copy-out semantics.

- SDA arguments
  A dummy SDA argument represents a (possibly undefined) reference to an SDA object. Its intent is IN and it is transferred according to the call-by-reference semantics.

- EVENT arguments
  A dummy EVENT argument represents a (possibly undefined) reference to an event. Its intent is IN and it is transferred according to the call-by-reference semantics.

Method dummy arguments may be distributed using the rules applicable to procedure arguments in HPF. Furthermore the HPF data parallel statements and directives may be used to explicitly express data parallelism.

The full definition of the SDA type *buffer_type* from the above example is given below. Here *fifo* is managed as a cyclic data structure implementing a FIFO discipline.

```
SDA TYPE buffer_type(size)
    !HPF$ PROCESSORS directives
    INTEGER :: size
    REAL :: fifo(0:size-1)
    INTEGER :: count=0
    INTEGER :: px=0, cx=0
    !HPF$ ALIGN & DISTRIBUTE directives
    ...
CONTAINS
    SUBROUTINE putb(x) WHEN (count .LT. size)
        REAL, INTENT(IN) :: x
        fifo(px) = x
        px = MOD(px+1,size)
        count = count + 1
    END

    SUBROUTINE getb(x) WHEN (count .GT. 0)
        REAL, INTENT(OUT) :: x
        x = fifo(cx)
        cx = MOD(cx+1,size)
        count = count - 1
    END
    ...
END buffer_type
```

2.3 SDA Declaration

2.3.1 Syntax

sda-declaration-stmt	**is**	*sda-type-spec* [[,*attr-spec(R503)*]... : :] *entity-decl-list(R504)*
sda-type-spec	**is**	`SDA` [(*sda-type-name*)]

Constraints:

1. `POINTER` attribute is not allowed

2.3.2 Semantics

An **SDA declaration** introduces one or more **SDA variables**. The declaration and all associated variables are called *typed*, if an SDA type name is specified in the declaration, otherwise *generic*. The syntax is similar to that of Fortran 90 type declaration statements referring to a derived type.

Assume that v is either an object declared in the entity declaration list, or, if the object is an array, an associated array element. Then v can be used to *create* an SDA object (cf. Section 3.5); after creation, v serves as a handle to that object.

For a generic declaration, v can be bound to an SDA object of arbitrary type. Otherwise, the specified parameterized SDA type and the object type must conform.

The following code fragment declares SDA variables S1, S2, and S3, all of which are associated with the parameterized SDA type *buffer_type*. S4 is a generic SDA variable.

SDA (buffer_type) :: S1
SDA (buffer_type) :: S2(n,m)
SDA (buffer_type), ALLOCATABLE, DIMENSION(:) :: S3
SDA S4

An SDA variable declaration is not allowed to have the **POINTER** attribute. Conceptually an SDA variable is a pointer to an SDA object. As a consequence, all tasks to which an SDA object is passed have access to the same copy of the object and hence can communicate with each other using the object.

An SDA variable can be an argument for procedures and methods.

Generic SDA Variables The creation of an SDA object in connection with a generic SDA variable is only possible by referring to an external SDA object (cf. Section 3.5): the new object inherits the type and state of the external object. Note that using this facility implies runtime checks to determine whether a method called with such an object exists and, if it does, whether the argument types match. However, a judicious use of status variables provides a graceful failure mode.

3 Method Activation

3.1 Syntax

method-activation	**is**	*s-task-activation*
	or	*a-task-activation*
s-task-activation	**is**	[CALL] *method-ref*
method-ref	**is**	*sda-variable*%*procedure-ref* [*status-clause*]
status-clause	**is**	, STAT = *stat-variable*
procedure-ref	**is**	*function-reference(R1209)*
	or	*subroutine-name* [([*actual-arg-spec-list(R1211)*])]
	or	*intrinsic-method-call*
a-task-activation	**is**	[*event-clause*] SPAWN *task-ref*
task-ref	**is**	*method-ref*
	or	*subroutine-name* [([*actual-arg-spec-list(R1211)*])]
		[*spawn-attr*][*status-clause*]
spawn-attr	**is**	ON(*resource-request*)
event-clause	**is**	*event-variable*=

Constraints:

1. A method-ref in a-task-activation must not contain a function-reference

3.2 Introduction

The successful activation of a method creates a **task**. Tasks are atomic in the sense that no two tasks associated with the same SDA object may be active at any given time. The task executing the method activation is called the *parent task*, the newly generated task the *child task*.

Depending on the calling conventions and on the resulting relationship between parent and child tasks, two kinds of method activations are distinguished:

3.3 Synchronous Method Activation

The synchronous method activation blocks the caller (parent task) until the callee (child task) finishes execution and returns. The syntax for synchronous activations is similar to that used for Fortran 90 subroutine and function calls. Function methods may be only activated in a synchronous way.

The values returned by the call to a function method can be utilized in the usual way after the call is completed. Similarly, the values returned for dummy method arguments with intent OUT or INOUT are well-defined once the child task has completed execution.

The example below illustrates how the methods defined for the buffer SDA introduced above may be synchronously activated in a program unit having access to two buffers.

```
...
SDA(buffer_type) :: a,b
REAL :: x
...
a%getb(x)  ! The first element of buffer "a" is retrieved and stored in x
b%putb(x)  ! Element "x" is stored in buffer "b"
           ! The following statement is executed after putb has finished its execution.
...
```

In this code the first element of buffer "a" is copied to buffer "b". The getb method has to be activated synchronously in order to avoid the activation of putb with an undefined argument. However, the execution of the parent task is also delayed by the synchronous call to putb, which might not be necessary in this example. We show below how the putb method can be activated asynchronously.

3.4 Asynchronous Method Activation

An asynchronous method activation is generated as a result of executing a *spawn* statement. The parent task continues its execution asynchronously with the child task and there is no implicit synchronization between them. Therefore any access to a variable passed to the child task with intent OUT or INOUT can result in a race condition. However, the language provides facilities for *explicit synchronization* as shown in Section 4.

The modification of our previous example illustrates how the putb method can be activated asynchronously, thus enabling the parallel execution of the parent task and the putb task. The keyword **CALL** preceding the getb activation may be omitted.

```
...
SDA(buffer_type) :: a,b
REAL :: x
...
CALL a%getb(x)
SPAWN b%putb(x)     ! putb is spawned as asynchronous task executing in
                    ! parallel with the caller.
...
```

Spawning of Procedures The full specification of an SDA may imply a significant amount of redundant notation if an SDA does not take advantage of the data sharing facilities. In order to eliminate some of this redundant overhad, OPUS allows procedures not declared within an SDA type to be directly spawned in the same manner as an asynchronous method activation. This means that there is no requirement in this case to explicitly define an associated SDA type and variable, and to explicitly create an SDA. All these actions are taken care of by the OPUS compiler. Tasks generated by spawning a procedure are referred to as **procedure tasks**.

Consider the specification of a producer generating data to be stored in our fifo buffer. The following code fragment outlines the required framework using the full SDA notation.

```
SDA TYPE producer_type
CONTAINS
  SUBROUTINE produce(b,...)
    SDA(buffer_type) b
      ...
  END
END SDA
  ...
SDA(producer_type) :: producer_variable
  ...
producer_variable%CREATE() ON(...)
  ...
SPAWN producer_variable%produce(...)
  ...
```

Since this code does not exploit the data sharing facilities of SDAs it contains a significant amount of redundant notation. Using the possibility to directly spawn procedures, most of this redundancy can be removed. The resources allocated to the SDA can be directly specified when spawning the procedure using the same **ON** clause. The code fragment below is semantically equivalent to the above program:

```
SUBROUTINE produce(b,...)
  SDA(buffer_type) b
    ...
END
  ...
SPAWN produce(...) ON(...)
  ...
```

3.5 Intrinsic Methods

3.5.1 Syntax

intrinsic-method-call	**is**	*sda-creation*
	or	*sda-termination*
	or	*sda-storage*
	or	*sda-access*
sda-creation	**is**	*create-stmt*
	or	*load-stmt*
create-stmt	**is**	`CREATE(`*actual-arg-spec-list(R1211)*`)` [*creation-attr*]
create-attr	**is**	`ON(`*resource-request*`)`
load-stmt	**is**	`LOAD(`*external-sda-identification*`)` [*creation-attr*]
external-sda-identification	**is**	*char-variable*
	or	*char-literal-constant(R420)*
sda-termination	**is**	`TERMINATE`
	or	`KILL`
sda-storage	**is**	`SAVE(`*external-sda-identification*`)`
sda-access	**is**	`GET(`*sda-component*`,` *variable(R601)*`)`
	or	`PUT(` *sda-component*`,` *expr(R723)*`)`
sda-component	**is**	*sda-structure*`%`*component-name*
sda-structure	**is**	*sda-variable-name*
	or	*sda-component*
resource-request	**is**	[*machine-request*,] [*processor-request*]
machine-request	**is**	`MACHINE=`*machine-spec*
machine-spec	**is**	*direct-mspec*
	or	*indirect-spec*
direct-mspec	**is**	*machine-name*
indirect-spec	**is**	*sda-variable*
	or	*task-specification*
	or	`SELF`
task-specification	**is**	*event-variable*
processor-request	**is**	`PROCESSORS=`*processor-spec*
processor-spec	**is**	[*allocation-strategy*] *processors-decl-list(H330)*
allocation-strategy	**is**	`NEW`
	or	`SAME`

3.5.2 Introduction

OPUS provides a set of intrinsic methods which are common to each SDA. These methods support the creation, saving, and termination of an SDA. Furthermore methods for accessing an SDA's internal data are provided.

3.5.3 Creating an SDA Object

Two methods for creating an SDA object are supported: The method *CREATE* is used to explicitly specify an object based on an SDA definition while *LOAD* constructs an SDA object by copying information from an external object which has been saved earlier. Some of the concepts which we will use here are discussed in the following sections.

The creation process takes typically five steps which might slightly differ for *CREATE* and *LOAD*. This will be discussed separately later.

Let v denote the SDA variable referred to by *CREATE* or *LOAD*. Then, based on the declaration of v and the arguments and attributes of the call, an object, B, is constructed by executing the following steps:

1. Determine an SDA type, t.
2. Perform resource allocation, as specified in the (optional) resource request.
3. Allocate data structures for B, as determined by t, the resource allocation, and distribution and alignment directives associated with t.
4. Initialize B.
5. Bind B to v.

The completion of the creation starts the *lifetime* of B. B is now accessible, via v, for activations of arbitrary methods associated with t, except for object creation. The lifetime of B terminates when *TERMINATE* or *KILL* (cf. Section 3.5.5) is activated.

If a creation fails for any reason, and a status clause has been specified, then the status variable is set to a non-zero value. Applying the *CREATE* method to a variable which already has been bound to an SDA object results in a runtime error.

CREATE The SDA variable, v, to which *CREATE* is applied, must be typed and must not be bound to an object. Let tt denote the associated parameterized SDA type. Steps 1 and 4 from above are performed as follows:

- Step 1:
  For all dummy arguments of tt, actual arguments must be provided in the create statement. These arguments are bound to the corresponding dummy arguments. As a result of this, SDA type t is obtained.
- Step 4:
  Default initialization of B is performed, as required for a well-defined initial state. After this, the (optional) user-specified initialization method (cf. 3.5.4) is called.

We illustrate the creation of a buffer SDA as introduced above.

```
SDA(buffer_type) :: S1, S2(N,M)       ! An SDA variable S1 associated with SDA type
                                      ! buffer_type and an SDA array S2 are declared.
                                      ! Their values are initially undefined.
...
CALL S1%CREATE(128)                   ! This method call creates an SDA object of type
                                      ! buffer_type with size 128.
                                      ! The SDA variable S1 refers to the new object.
DO I=1,N
  DO J=1,M
    CALL S2(I,J)%CREATE(32*MAX(I,J))    ! This statement causes the creation
                                 ! of N*M SDA objects of type buffer_type
                                 !  with the size depending on I and J.
...
```

LOAD The SDA variable v may either be typed or generic, with type attribute *tt*. It may not be bound to an object. Steps 1 and 4 from above are performed as follows:

- Step 1:
  The load statement specifies (by means of a variable or a constant) a character string which identifies an external SDA object, x. Let t_x denote the SDA type of x. t_x must conform to *tt* (trivial if *tt* is of type *generic*). t is defined to be t_x.

- Step 4:
  The data structures of B are initialized by copying out the values of x.

Resource Request Each SDA operates on a set of resources which are allocated at the time the SDA object is created, and deallocated at the time of its termination.

Different SDA objects may execute on disjoint or overlapping sets of resources. Note that the resource request as discussed below applies also for procedure tasks.

A resource request may consist of the *machine specification* and the *processor specification.* The *machine specification* identifies the (physical or logical) machine on which the SDA object is created. The machine can be specified either *directly* or *indirectly.* A *direct specification* identifies a machine by a string with a system-dependent meaning. An *indirect specification* provides the identification of an SDA (which must have been initialized) or of a procedure task (which must be active). Indirect machine specification associates the machine allocated to the given SDA with the newly created SDA. A procedure task is identified by its event variable. Note that an indirect machine specification using event variables is only possible when referring to procedure tasks, since they are not associated with an SDA. For the ease of programming the intrinsic attribute *SELF* identifies the resources allocated to the task creating the new SDA. *SELF* is also the default value used if the machine specification is omitted.

The *processor specification* can only be given directly. If it is omitted either all processors available on the specified machine (in case of direct machine specification) or the processors assigned to the given SDA (in case of indirect machine specification) are devoted to the newly created SDA object. The direct processor specification uses the syntax and semantics of the HPF-processor declaration. It might be preceded by the keyword **NEW** or **SAME** if the machine specification was done indirectly. If **NEW** is used, then the specified processors are allocated in addition to those already associated with the given SDA; in contrary, **SAME** is used to ensure that exactly the processors associated with the given SDA are allocated. If no allocation strategy is specified, the allocation of the processors is system dependent and they might be overlapping with those associated with the other SDA.

We illustrate some possibilities for an resource request using the buffer SDA introduced above.

SDA(buffer_type) :: a,b
...

- a%*CREATE*(128)
  ! The SDA is created on the same resources allocated to the parent task

- a%*CREATE*(128) **ON MACHINE**='Meiko'
  ! The SDA is created on the machine 'Meiko' using all available processors.

- b%*CREATE*(128) **ON MACHINE**=a, **NEW PROCESSORS**=32
  ! The SDA is created on the same machine as SDA 'a', using 32 processors
  ! which are disjoint from the processors associated with 'a'.

- b%*CREATE*(128) **PROCESSORS**=32
  ! The SDA is created on the same machine associated with the parent task
  ! using 32 processors which might overlap with the processors allocated to the caller.

3.5.4 Initialization of an SDA Object

After the creation of an SDA object its internal variables are default initialized as required for a well-defined initial state. In addition to the default initialization the user can specify a subroutine method to be an *initialization method* (similar to C++ *constructors* [4]). The name of the initialization method must be the same as the SDA type name and the method must not yield any return value. The user-supplied initialization method is implicitly executed immediately after the object creation.

Syntactically, initialization methods are treated as normal subroutine methods, thus no additional syntax is required.

3.5.5 Termination of an SDA Object

The intrinsic method *TERMINATE* is used to terminate the lifetime of an SDA object. When calling *TERMINATE* the SDA variable must be bound to an object. The effect of this method activation is the deletion of the object.

If execution leaves the scope of the program unit in which an SDA declaration was processed, an implicit *TERMINATE* is executed.

The activation of *TERMINATE* is secure in the sense that it is ensured that all child tasks as well as enqueued method calls must have finished their execution. Only if no active child task exists for an SDA, *TERMINATE* will cause the termination of all created SDAs and delete the object.

A more brute force alternative to *TERMINATE* is the intrinsic method *KILL*. An activation of *KILL* causes the immediate termination of the lifetime of an SDA object and its deletion. The execution of all child tasks is stopped and all created SDAs are killed as well.

3.5.6 Saving an SDA Object

OPUS provides facilities to make SDA objects persistent by saving them on external storage. An external name for the saved object might be supplied to the intrinsic method *SAVE*. The name might be omitted when the object was loaded from external storage. In this case the external name specified in the loading method is used. If an object with the specified name does already exist, it is replaced with the object executing the method request. Otherwise a new object is created on the external storage.

When saving an object an exact image of the object's state in the time the method request is executed will be generated.

3.5.7 Access to an SDA's Data

Two intrinsic methods are provided for manipulating the data on an SDA. *PUT* is used to *generate* an SDA's internal variable, i.e. to assign it a value. Using the method *GET* an internal variable can be *consumed*, i.e. its value is retrieved. The syntax of *GET* and *PUT* is modeled after the Fortran 90 syntax for accessing derived types.

Each access using the standard Fortran 90 syntax for accessing a module's data is implicitly transformed to a *GET* or *PUT* method, respectively.

4 Synchronization

4.1 SDA Synchronization

As discussed in Section 2.2, SDAs provide implicit and explicit synchronization facilities to ensure a coherent state of their data. Implicit synchronization is achieved by allowing only one method of an SDA to be active at a time. This method has exclusive access to the whole SDA and is an atomic operation. This feature is similar to monitors in shared memory languages.

The possibility of assigning a condition clause (the **WHEN** construct) to an SDA's method allows explicit synchronization. A method may only be activated when its associated condition clause evaluates to TRUE.

4.2 Task Synchronization

4.2.1 Syntax

wait-stmt	**is**	`WAIT(`*event-variable*`)` [*status-clause*]
	or	`WAITANY(`*event-variable-list*`)` [*status-clause*]
test-stmt	**is**	`TEST(`*event-variable*`)` [*status-clause*]

4.2.2 Semantics

Asynchronously executing tasks do not provide any implicit synchronization. This means that e.g. the parent task could access variables passed to a child task with intent OUT or INOUT before the child finishes its execution. This access can result in a race condition.

The language provides *event variables* to explicitly synchronize asynchronous tasks. A new type **EVENT** is introduced and all variables of that type must be explicitly declared. When a declaration of an event variable, say E, is processed, then E is implicitly set to *undefined*. No explicit initialization is possible in the declaration.

An asynchronous task is identified by an *event value* which has a *completion state*. This completion state determines whether or not the task has completed execution and is set to **false** at the creation. If an event clause with event variable E is specified as part of the spawn statement, E is bound to the task's completion state. E must be *undefined* when spawning the task.

An event variable can be accessed using the **WAIT**, **WAITANY**, or **TEST** statement.

The event variable E remains bound to the task's completion state, until all synchronization statements referring to E have completed their execution. Then E is set to *undefined* and can be used again. Similarly, a task's *event value* has to be active while there exists any event variable bound to it.

Wait The execution of a wait statement, **WAIT**(E) blocks the executing task until the method associated with E has finished its execution. If E is not bound to an event value, thus its value is *undefined*, the wait statement terminates unsuccessfully. If the method execution is already complete at the time the wait statement is executed, then its effect is empty.

At the time a wait statement is successfully completed, it is guaranteed that the values of all dummy arguments with intent OUT or INOUT of the associated method execution are updated.

Similar to **WAIT** the execution of **WAITANY**(*event-variable-list*) blocks the executing task until *any* method associated with $E \in$ *event-variable-list* has finished its execution. The position of E in the *event-variable-list* is returned.

The synchronization needed for the copy between buffers illustrated in Section 3 can also be achieved by using **WAIT** as shown below.

```
...
SDA(buffer_type) :: a,b
REAL :: x
EVENT :: e
...
e=SPAWN a%getb(x)
WAIT(e)                  ! Block execution until the getb-method has finished its execution
SPAWN b%putb(x)
...
```

Test A nonblocking alternative to the wait statement is the test statement, **TEST**(E). The execution of the test statement returns the actual value of the event value, to which E is bound, thus allowing the caller to continue execution. If the value of E is *undefined*, the test statement terminates unsuccessfully.

References

[1] B. Chapman, M. Haines, P. Mehrotra, J. Van Rosendale, and H. Zima. OPUS: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 1997.

[2] High Performance Fortran Forum. High Performance Fortran Language Specification Version 2.0, January 1997.

[3] ISO. Fortran 90 Standard. ISO/IEC 1539 :1991 (E).

[4] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, second edition, 1991.

Appendix

A Examples

A.1 Barrier Synchronization

Assume that N tasks are to be synchronized at a barrier, where N can be determined at runtime. This can be done by introducing SDA type *barrier*, as specified below. One complication in this algorithm is caused by the fact that tasks leave the barrier sequentially. We have to avoid that an already released task reaches the barrier again, while there is still another task waiting for release. This is the reason why registering is only allowed if $count < N$.

```
SDA TYPE barrier_type(N)
  INTEGER :: N                          ! number of tasks
  INTEGER, PRIVATE :: count=0           ! number of tasks reaching the barrier
  INTEGER, PRIVATE :: r=0               ! number of tasks passing the barrier
CONTAINS

  SUBROUTINE register WHEN (count .LT. N)
    count = count + 1
  END

  SUBROUTINE test WHEN (count .EQ. N)
    r = r + 1
    IF (r .EQ. N) count = 0; r = 0
  END
  ...
END barrier_type


SDA(barrier_type) :: barrier
INTEGER :: num_tasks
 ...
 compute num_tasks
 CALL barrier%CREATE(num_tasks)
 generate num_tasks tasks
 ...
```

In each task, access to the barrier is represented by the following sequence of method calls:

```
SUBROUTINE XXX(b)
 SDA(barrier_type) :: b
 ...
 CALL b%register
 CALL b%test
 ...
END XXX
```

A.2 Exchange of Signals Between Tasks

In this section, we illustrate how tasks can send signals to each other using SDAs. Assume that there exist two subroutines T_1 and T_2, each being spawned as one asynchronous task. Both tasks want to inform each other about the progress in their execution. Under certain circumstances, a task may have to wait until the other task has reached a given program point.

Let $L_1(i), 1 \leq i \leq N$ and $L_2(i), 1 \leq i \leq N$[1] denote the sets of labels in T_1 and T_2, respectively. We specify SDA type *signal* for coordinating the two tasks:

```
SDA TYPE signal(K,N)
  INTEGER :: K,N
  LOGICAL, PRIVATE :: L(K,N)= .FALSE.     ! Signals for K tasks with N labels
CONTAINS

  SUBROUTINE reaching(k,l)                ! task T_k reaches label L_k(l)
    L(k,l) = .TRUE.
  END

  SUBROUTINE wait_for(k,l) WHEN (L(k,l))
  END
END signal
```

Assume s to be an SDA variable associated with *signal*. If task T_1 reaches label $L_1(l)$, it executes **CALL** $s\%reaching(1,l)$. If task T_1 must wait until the other task T_2 reaches $L_2(l)$, then it executes **CALL** $s\%wait\_for(2,l)$.

A.3 Consumer/Producer Problem

The **consumer/producer problem** represents a class of coordination problems in which one or more independent *producer* tasks compute some values, and one or more independent *consumer* tasks read these values and use them for further processing. The details of producing and consuming are irrelevant for our discussion and ignored. The analysis of the problem focuses on how producers and consumers are coordinated. The assumption made here is that there exists a *bounded buffer* into which all producers write and from which all consumers read. The buffer is managed as a cyclic memory using a FIFO strategy.

The consumer/producer problem is one of the basic coordination problems that occur in practice in many different variants.

More specifically, we formulate the problem as follows:

- Consider a bounded buffer as specified by the SDA type *buffer_type* of Figure 1. The buffer is organized as a FIFO queue; writing into the buffer (method *putb*) and reading from the buffer (method *getb*) are atomic actions. In the initial state, all elements of the buffer are *empty*. A call to *putb* consumes one empty element and creates one *full* element; similarly, a call to *getb* consumes one full element and creates one empty element. The condition clauses associated with these two methods implement the required synchronization conditions: (1)

[1] L_1, L_2 have the same range for reasons of convenience, without restricting generality.

a producer may write into the buffer only if there exists at least one empty element, and (2) a consumer may read from the buffer only if there exists at least one full element.

- There are $nc \geq 1$ consumer tasks $C(i), 1 \leq i \leq nc$. Each $C(i)$ is a cyclic task, in each iteration reading the next full element from the buffer and then consuming it.
- There are $np \geq 1$ producer tasks, $P(i), 1 \leq i \leq np$. Each $P(i)$ is a cyclic task, in each iteration producing an element and then writing it into the buffer.
- Apart from the synchronization explained above, all consumer and producer tasks may work asynchronously in parallel.

A solution to the problem is given in Figure 1.

A.4 Readers/Writers Problem

Another common class of coordination problems is represented by the **readers/writers problem**. It is characterized by a set of one or more independent *readers* and a set of one or more independent *writers* each of which is accessing a common data resource. Writing is exclusive while reading can be done by more than one task in parallel. Moreover, deadlock and starvation must be excluded.

We require the following synchronization conditions to be satisfied:

1. Reader and writer tasks cannot access data resource D at the same time.
2. At most one writer task can access D at any time.
3. If, at the time a writer task terminates execution, there is at least one registered reader task, then all readers registered at this time are given access to the resource.
4. Apart from the previous case, a reader gains access to the resource only if there are no registered writers.

The last two conditions are required to avoid starvation.

We will not go into the details of SDA types, variable and SDA creation for reader and writer tasks, since this can be done along the same lines as for the producers and consumers of the previous example. However, we will discuss the details of the scheduling strategy.

The algorithm is complicated by the fact that starvation must be excluded. The code executed in each reader task has the following structure (the SDA *read_write* is specified in Figure 2):

```
SUBROUTINE read(d)
  SDA(read_write) d
  ...
  DO
    ...
    CALL d%register_read
    CALL d%begin_read
    read access to resource
    CALL d%end_read
    ...
  END DO
END read
```

```
PROGRAM Consumer_Producer
SDA TYPE buffer_type(size)
  INTEGER :: size
  REAL, PRIVATE :: fifo(0:size-1)
  INTEGER, PRIVATE :: count=0          ! number of full elements in fifo
  INTEGER, PRIVATE :: px=0             ! producer index
  INTEGER, PRIVATE :: cx=0             ! consumer index
CONTAINS
  SUBROUTINE putb(x) WHEN (count .LT. size)
    REAL, INTENT(IN) :: x
    fifo(px) = x                       ! Put x into first empty buffer element
    px = MOD(px+1,size); count = count + 1
  END
  SUBROUTINE getb(x) WHEN (count .GT. 0)
    REAL, INTENT(OUT) :: x
     x = fifo(cx)                      ! Read next full buffer element
     cx = MOD(cx+1,size); count = count - 1
  END
  ...
END buffer_type

SUBROUTINE produce(b)
  SDA(buffer_type) b
  ...
  DO
    produce a data item A
    CALL b%putb(A)
  END DO
END produce

SUBROUTINE consume(b)
  SDA(buffer_type) b
  ...
  DO
    CALL b%getb(A)
    consume A
  END DO
END consume

INTEGER :: np, nc, buffersize
SDA(buffer_type) :: buffer
READ(np,nc,buffersize)
CALL buffer%CREATE(buffersize)         ! Create buffer SDA
DO i=1,np                              ! Spawn producer tasks
  SPAWN produce(buffer)
END DO
DO i=1,nc                              ! Spawn consumer tasks
  SPAWN consume(buffer)
END DO
END PROGRAM
```

Figure 1: Consumer/Producer Problem

Similarly, the code executed in each writer task is of the form:

```
SUBROUTINE write(d)
  SDA(read_write) d
  ...
  DO
    ...
    CALL d%register_write
    CALL d%begin_write
    write access to resource
    CALL d%end_write
    ...
  END DO
END write
```

Note that access to the resource in readers as well as writers must go through a two-step process. The first step is *registration*: the task simply registers (as a reader or writer, respectively), thus indicating its plan to access the data resource immediately following registration. No blocking can occur in this step. The second step represents the actual entry into the region in which access is performed. This is protected by a method condition guaranteeing the synchronization conditions specified above.

We will use the following terminology:

- A reader (writer) is *registered* for access to the resource if it has passed registration but not yet completed *begin_read* (or *begin_write*, respectively). The number of registered readers (writers) will respectively be stored in the variables *rr* and *rw*.
- A reader (writer) is *accessing* the resource if it has passed registration and *begin_read* (or *begin_write*, respectively), but has not yet executed *end_read* (or *end_write*, respectively). The number of accessing readers (writers) will respectively be stored in the variables *ar* and *aw*.
- If the situation described under synchronization condition 3 occurs, a writer ending access must release all readers that are registered at this time. This is called the set of *admitted readers*. No writer is allowed to access the resource until all admitted readers have gained access. This is handled via the auxiliary variables *admitted_readers* and *reader_release* and assumes that there is a FIFO strategy for handling a queue of blocked tasks with respect to any specific method call.

The full SDA specification is given in Figure 2.

```
SDA TYPE read_write
  INTEGER, PRIVATE  rr=0                    ! number of registered readers
  INTEGER, PRIVATE  rw=0                    ! number of registered writers
  INTEGER, PRIVATE  ar=0                    ! number of accessing readers
  INTEGER, PRIVATE  aw=0                    ! number of accessing writers
  INTEGER, PRIVATE  admitted_readers
  LOGICAL, PRIVATE  reader_release=.FALSE.
  ...
CONTAINS
  SUBROUTINE register_read
    rr = rr+1
  END
  SUBROUTINE begin_read WHEN(aw .EQ. 0 .AND. &
    & (rw .EQ. 0 .OR. reader_release))
    ar = ar+1
    rr = rr-1
    IF (reader_release) THEN
      admitted_readers=admitted_readers-1
      IF (admitted_readers .EQ. 0) reader_release=.FALSE.
    ENDIF
  END
  SUBROUTINE end_read
    ar = ar-1
  END
  SUBROUTINE register_write
    rw = rw+1
  END
  SUBROUTINE begin_write WHEN(aw .EQ. 0 .AND. &
    & (ar .EQ. 0 .AND. .NOT. reader_release))
    aw = 1
    rw = rw-1
  END
  SUBROUTINE end_write
    aw = 0
    IF (rr .GT. 0) THEN
      admitted_readers = rr
      reader_release = .TRUE.
    ENDIF
  END
END read_write
```

Figure 2: Readers/Writers problem

B Syntax

Notation

The notation used in the syntax rules is a variation of Backus-Naur form and in accordance with the Fortran 90 standard document [3].

The syntactic metasymbols used are:

is	introduces a syntactic class definition
or	introduces a syntactic class alternative
[]	encloses an optional item
[]...	encloses an optionally repeated item which may occur zero or more times

Nonterminal symbols which are part of Fortran 90 are implicitly defined according to the Fortran 90 language specification [3] and annotated with the Fortran 90 rule number (prefix *R*). Nonterminal symbols which are part of High Performance Fortran [2] are accordingly annotated with the HPF rule number (prefix *H*).

Example:

sda-specification-part	**is**	*specification-part(R204)*
processor-request	**is**	`PROCESSORS=`*processors-decl-list(H330)*

Assumed syntax rules are:

xyz-list	**is**	*xyz* [, *xyz*] ...
xyz-name	**is**	*name(R304)*
integer-xyz	**is**	*xyz*
xyz-variable	**is**	*variable(R601)*

B.1 SDA Types

sda-type	**is**	*sda-stmt* [*sda-specification-part*] [*sda-method-part*] *end-sda-stmt*
sda-stmt	**is**	`SDA TYPE` *type-name* [`(`*dummy-arg-list(R1221)*`)`]
sda-specification-part	**is**	*specification-part(R204)*
sda-method-part	**is**	*contains-stmt(R1225) method* [*method*]...
method	**is**	*function-method*
	or	*subroutine-method*
function-method	**is**	*function-stmt(R1216)* [*condition-clause*] *subprogram-body* *end-function-stmt(R1218)*
condition-clause	**is**	`WHEN (`*scalar-logical-expr(R725)*`)`
subprogram-body	**is**	[*specification-part(R204)*] [*execution-part(R208)*] [*internal-subprogram-part(R210)*]
subroutine-method	**is**	*subroutine-stmt(R1220)* [*condition-clause*] *subprogram-body* *end-subroutine-stmt(R1222)*
end-sda-stmt	**is**	`END` [`SDA TYPE` [*type-name*]]

B.2 SDA Declaration

sda-declaration-stmt	**is**	*sda-type-spec* [[,*attr-spec(R503)*]... `::`] *entity-decl-list(R504)*
sda-type-spec	**is**	`SDA` [`(`*sda-type-name*`)`]

Constraints:

1. `POINTER` attribute is not allowed

B.3 Method Activation

method-activation	**is**	*s-task-activation*
	or	*a-task-activation*
s-task-activation	**is**	[`CALL`] *method-ref*
method-ref	**is**	*sda-variable*`%`*procedure-ref* [*status-clause*]
status-clause	**is**	, `STAT =` *stat-variable*
procedure-ref	**is**	*function-reference(R1209)*
	or	*subroutine-name* [`(`[*actual-arg-spec-list(R1211)*]`)`]
	or	*intrinsic-method-call*
a-task-activation	**is**	[*event-clause*] `SPAWN` *task-ref*
task-ref	**is**	*method-ref*
	or	*subroutine-name* [`(`[*actual-arg-spec-list(R1211)*]`)`] [*spawn-attr*][*status-clause*]
spawn-attr	**is**	`ON(`*resource-request*`)`
event-clause	**is**	*event-variable*`=`

Constraints:

1. A method-ref in a-task-activation must not contain a function-reference

B.4 Intrinsic Methods

intrinsic-method-call	**is**	*sda-creation*
	or	*sda-termination*
	or	*sda-storage*
	or	*sda-access*

Creating, Saving, and Terminating SDA Objects

sda-creation	**is**	*create-stmt*
	or	*load-stmt*
create-stmt	**is**	`CREATE(`*actual-arg-spec-list(R1211)*`)` [*creation-attr*]
create-attr	**is**	`ON(`*resource-request*`)`
load-stmt	**is**	`LOAD(`*external-sda-identification*`)` [*creation-attr*]
external-sda-identification	**is**	*char-variable*
	or	*char-literal-constant(R420)*
sda-termination	**is**	`TERMINATE`
	or	`KILL`
sda-storage	**is**	`SAVE(`*external-sda-identification*`)`
resource-request	**is**	[*machine-request*,] [*processor-request*]
machine-request	**is**	`MACHINE=`*machine-spec*
machine-spec	**is**	*direct-mspec*
	or	*indirect-spec*
direct-mspec	**is**	*machine-name*
indirect-spec	**is**	*sda-variable*
	or	*task-specification*
	or	`SELF`
task-specification	**is**	*event-variable*
processor-request	**is**	`PROCESSORS=`*processor-spec*
processor-spec	**is**	[*allocation-strategy*] *processors-decl-list(H330)*
allocation-strategy	**is**	`NEW`
	or	`SAME`

Access to SDA data

sda-access	**is**	`GET(` *sda-component* , *variable(R601)* `)`
	or	`PUT(` *sda-component* , *expr(R723)* `)`
sda-component	**is**	*sda-structure* `%` *component-name*
sda-structure	**is**	*sda-variable-name*
	or	*sda-component*

B.5 Task Synchronization

wait-stmt	**is**	`WAIT(`*event-variable*`)` [*status-clause*]
	or	`WAITANY(`*event-variable-list*`)` [*status-clause*]
test-stmt	**is**	`TEST(`*event-variable*`)` [*status-clause*]