3. **Subprogram access and Polychrony**

11.3.2 Subprogram group access

1. **Abstract syntax of *Subprogram_group_access***

$$Subprogram\_group\_access ::= subprogram\_group\_access\_ID \times Access\_status \times \mathbf{opt}(Subprogram\_group\_reference) \times \mathbf{opt}(\mathbf{list}(Subprogram\_group\_access\_property))$$
$$Subprogram\_group\_reference ::= subprogram\_group\_ID$$

2. **Standard properties**

Property	Provides	Requires
Allowed_Connection_Binding_Class	X	X
Allowed_Connection_Binding	X	X
Actual_Connection_Binding	X	X
Required_Connection	X	X
Acceptable_Array_Size	X	X

3. **Subprogram group access and Polychrony**

11.4 Data access

Data access is used to model shared data.

Components can declare that they require access to externally declared data components. Components may provide access to their data components.

11.4.1 Abstract syntax of *Data access*

A requires data access declaration indicates that a component requires access to a component declared external to the component. For data components, different forms of required access, such as read-only access, are specified by a Access_Right property.

A provides data access declaration indicates that a subcomponent provides access to a data component contained in the component.

Shared data may be accessed by multiple threads. Such potential concurrent access is controlled according to the Concurrency_Control_Protocol property. This property applies to data.

$$Data\_access ::= data\_access\_ID \times Access\_status \times \mathbf{opt}(Data\_reference) \times \mathbf{opt}(\mathbf{list}(Data\_access\_property))$$
$$Data\_reference ::= dataID$$

Figure 20 shows two types of data access. *Data1* is made accessible outside *Thread1* through a *provides data access* feature declaration of *Thread1*. It is being accessed by *Thread2* as expressed by a *requires data access* feature declaration in the thread type of *Thread2*.

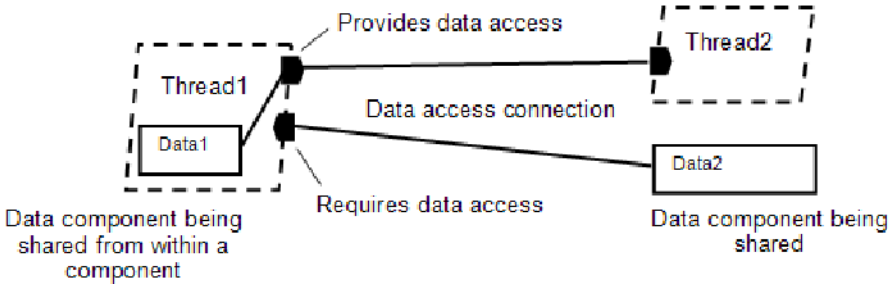

Figure 20: Data access

```
thread Thread1
  features
    Dataset: provides data access Data1;
end Thread1;

thread Thread2
  features
    Reqdataset: requires data access Data1;
end Thread2;
```

11.4.2 Standard properties

Property	Provides	Requires
Access_Right	X	X
Access_Time	X	X
Base_Address	X	X
Acceptable_Array_Size	X	X

The **Access_Time** property specifies the time range over which a component has access to a shared data component. By default, access is required for the duration of the component execution. The value of a shared data component is read or written through the use of a data variable that represents the shared data component, or through Get_Value and Put_Value service calls. Write access immediately updates the shared data component.

Access_Time : **record** (First: IO_Time_Spec; Last: IO_Time_Spec;)
⇒ (First ⇒ (Time ⇒ Start; Offset ⇒ 0.0ns .. 0.0 ns;);
Last ⇒ (Time ⇒ Completion; Offset ⇒ 0.0ns .. 0.0ns;);)
applies to (data access);

Between First time and Last time, the data component is being accessed. First (Last) is specified by a reference time and a time offset range. It is modeled as a constraint

(Between()).

$$
\begin{aligned}
&Access\_Time \ ::= \ First \times Last \\
&First \ ::= \ IO\_Time\_Spec \\
&Last \ ::= \ IO\_Time\_Spec \\
&IO\_Time\_Spec \ ::= \ TimeRange \times IO\_Reference\_Time \\
&TimeRange \ ::= \ Time \times Time \\
&Time \ ::= \ aadlinteger \times Time\_Unit
\end{aligned}
$$

11.4.3 Data access and Polychrony

1. **Requires data access**

   In Figure 21, two constraints are added. They represent the *First* and *Last* access time specified by **Access_Time** property. Each access time is represented by a renference time (*Time*) and an *Offset*.

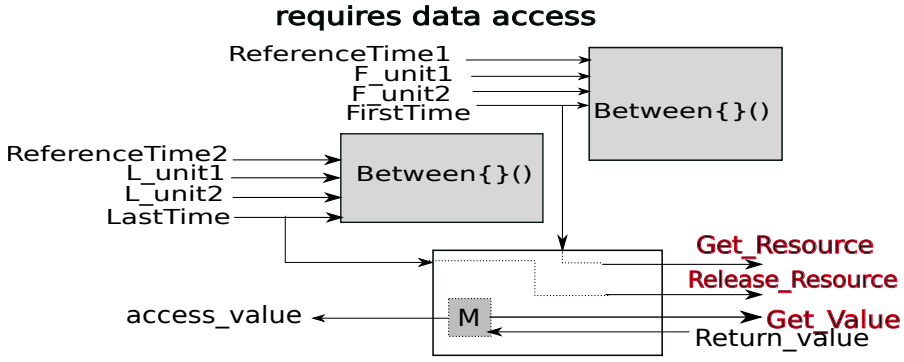

Figure 21: Requires data access

Get_Value, Get_Resource and *Release_Resource* are three predefined service calls. At *FirstTime*, a *Get_Resource* is performed to lock the data resource. At *LastTime*, a *Release_Resource* is sent out to unlock the resource.

A *Get_Value* may be sent to the corresponding data resource during the execution depending on the detailed implementation. The required value is returned (*Return_value*) and stored in a memory *M*. The memory is updated when a new *Get_Value* is performed (a new value is returned). The output *access_value* is the current value in the memory.

Let $x = (x_1, x_2, x_3) \in Requires\_data\_access$

where $x_1 \in data\_access\_ID$

$x_2 \in Data\_reference$

$x_3 = \{x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}, x_{37}, x_{38}, x_{39}, x_{30}, \} \in Access\_Time$

$x_{31}, x_{33}, x_{36}, x_{38} \in aadlinteger$

$x_{32}, x_{34}, x_{37}, x_{39} \in Time\_Unit$

$x_{35}, x_{310} \in IO\_Reference\_Time$

RequiresDataAccessTranslation$(x, \mathscr{C})$ =

$x'_1 :: access\_value := RequiresDataAccessFm, Fn, Lm, Ln$

$(FirstTime, F\_Reference\_Time, F\_unit1, F\_unit2,$

$LastTime, L\_Reference\_Time, L\_unit1, L\_unit2, DataResource)$

where $FirstTime, F\_Reference\_Time, F\_unit1, F\_unit2, LastTime,$

$L\_Reference\_Time, L\_unit1, L\_unit2, DataResource$ come from the context $\mathscr{C}$

Fm = **PropertyTranslation**(x_{31})

Fn = **PropertyTranslation**(x_{33})

F_unit1 = **PropertyTranslation**(x_{32})

F_unit2 = **PropertyTranslation**(x_{34})

$F\_ReferenceTime$ = **PropertyTranslation**(x_{35})

Lm = **PropertyTranslation**(x_{36})

Ln = **PropertyTranslation**(x_{38})

L_unit1 = **PropertyTranslation**(x_{37})

L_unit2 = **PropertyTranslation**(x_{39})

$L\_ReferenceTime$ = **PropertyTranslation**(x_{310})

$DataResource$ = **IDTranslation**(x_2)

$Data_type$ = **DataReferenceTranslation**(x_2)

The process RequiresDataAccess() is defined in library AADL_DATAACCESS.

```
process RequiresDataAccess =
{ integer F_min_offset, F_max_offset, L_min_offset, L_max_offset;}
(? event FirstTime, F_Reference_Time, F_unit1, F_unit2;
   event LastTime, L_Reference_Time, L_unit1, L_unit2;
   DataResource;
 ! Data_type access_value;)
  (| Between{F_min_offset, F_max_offset}
```

```
          (FirstTime, F_Reference_Time, F_unit1, F_unit2)
| Get_Resource(FirstTime, DataResource)
| return_value := Get_Value(DataResource)
| access_value := M(return_value, ^access_value)
| Between{L_min_offset, L_max_offset}
          (LastTime, L_Reference_Time, L_unit1, L_unit2)
| Release_Resource(LastTime, DataResource)
| )
where
  Data_type return_value
end;
```

Get_Resource(), *Release_Resource*() and *Get_Value*() are defined in a library. **Access_Time** property translation is explained in Section 14.4.

```
process Get_Resource =
(? event time; Resource;)

process Release_Resource =
(? event time; Resource;)

process Get_Value =
(? Resource; ! ReturnValue;)
```

2. **Provides data access**

   At *FirstTime*, a *Get_Resource* service call is performed. At *LastTime*, a *Release_Resource* service call is performed. A *Put_Value* service call is used to write a value to the data resource. A memory M receives values from associated thread or process. (Figure 22)

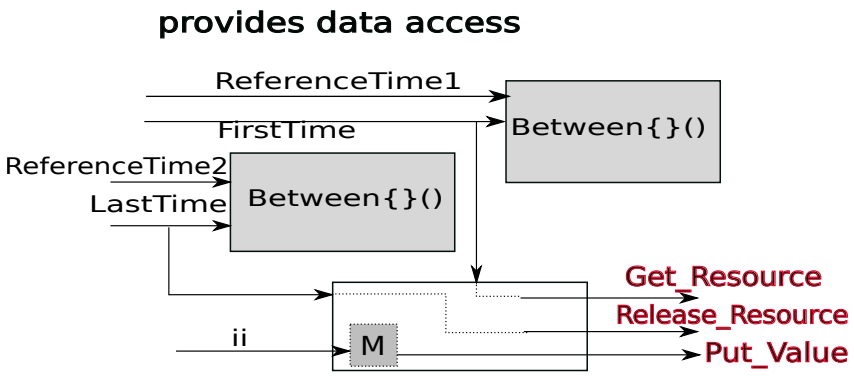

Figure 22: Provides data access

Let $x = (x_1, x_2, x_3) \in Provides\_data\_access$
where $x_1 \in data\_access\_ID$
$x_2 \in Data\_reference$
$x_3 = Access\_Time \in Data\_access\_property$

ProvidesDataAccessTranslation$(x, \mathscr{C})$ =
$x'_1 :: RrovidesDataAccessFm, Fn, Lm, Ln$
$(FirstTime, F\_Reference\_Time, F\_unit1, F\_unit2,$
$LastTime, L\_Reference\_Time, L\_unit1, L\_unit2, DataResource, value)$
where $FirstTime, F\_Reference\_Time, F\_unit1, F\_unit2, LastTime,$
$L\_Reference\_Time, L\_unit1, L\_unit2, DataResource, value$ come from the context $\mathscr{C}$
Fm = **PropertyTranslation**(x_{31})
Fn = **PropertyTranslation**(x_{33})
F_unit1 = **PropertyTranslation**(x_{32})
F_unit2 = **PropertyTranslation**(x_{34})
$F\_ReferenceTime$ = **PropertyTranslation**(x_{35})
Lm = **PropertyTranslation**(x_{36})
Ln = **PropertyTranslation**(x_{38})
L_unit1 = **PropertyTranslation**(x_{37})
L_unit2 = **PropertyTranslation**(x_{39})
$L\_ReferenceTime$ = **PropertyTranslation**(x_{310})
$DataResource$ = **IDTranslation**(x_2)
$Data_type$ = **DataReferenceTranslation**(x_2)

The process RrovidesDataAccess() is defined in library AADL_DATAACCESS.

```
process ProvidessDataAccess =
{ integer F_min_offset, F_max_offset, L_min_offset, L_max_offset;}
(? event FirstTime, F_Reference_Time, F_unit1, F_unit2;
   event LastTime, L_Reference_Time, L_unit1, L_unit2;
   DataResource; Data_type value;)
  (| Between{F_min_offset, F_max_offset}
           (FirstTime, F_Reference_Time, F_unit1, F_unit2)
   | Get_Resource(FirstTime, DataResource)
   | Put_Value(DataResource, pvalue)
   | pvalue := M(value, ^pvalue)
   | Between{L_min_offset, L_max_offset}
           (LastTime, L_Reference_Time, L_unit1, L_unit2)
```

```
    | Release_Resource(LastTime, DataResource)
    | )
    where
      Data_type pvalue
    end;

process Put_Value =
(? Resource; Value;)
```

11.5 Bus access

Bus components can be made accessible to other components. Components can declare that they require access to externally declared buses. Components may provide access to their buses. Bus access is used to model connectivity of execution platform components through buses.

A requires bus component access declaration indicates that a component requires access to a component declared external to the component. Required bus accesses are resolved to actual bus subcomponents through access connection declarations.

A provides bus access declaration indicates that a subcomponent provides access to a bus contained in the component. Provided bus accesses can be used to resolve required bus access.

A bus that is accessed by more than one component is shared. The shared bus is a common resource through which processor, memory and device components communicate.

11.5.1 Abstract syntax of *Bus access*

$$
\begin{aligned}
&Bus\_access ::= bus\_access\_ID \times Access\_status \times \\
&\quad \mathbf{opt}(Bus\_reference) \times \mathbf{opt}(\mathbf{list}(Bus\_access\_property)) \\
&Bus\_reference ::= busID
\end{aligned}
$$

11.5.2 Standard properties

Property	Provides	Requires
Access_Right	X	X
Acceptable_Array_Size	X	X

11.5.3 Bus access and Polychrony

11.6 Feature group

Feature groups represent groups of component features of feature groups.

The content of a feature group is declared through a feature group type declaration. This declaration is then referenced when feature groups are declared as component features.

Within a component, the features of a feature group can be connected to individually. Outside a component, feature groups can be connected as a single unit.