Seuss for Java

Language Reference

Rajeev Joshi [1]

12 February 1998

Abstract

The programming model *Seuss* comprises of (i) a notation for writing concurrent programs, (ii) a logic for proving program properties and (iii) an operational semantics for assigning a meaning to program executions. Seuss is based on the observation that concurrent programs typically consist of large amounts of sequential code, which are often written, understood, and reasoned about in isolation, while concurrency is made explicit only at the highest level, in deciding how to orchestrate the executions of these sequential programs. Consequently, the Seuss notation, in its most abstract form [Misra 96], describes structuring constructs for organising sequential programs written in any programming language. This document describes the language *Seuss for Java* , which is an adaptation of the original Seuss notation for use with the programming language *Java* [GJS 96]. The document provides a syntax in extended Backus-Naur form (EBNF) and an operational semantics for the language. It is intended for programmers with a knowledge of Java; thus, although *Seuss for Java* programs contain sections of Java code, details of the Java syntax (e.g., the shape of expressions, class declarations, packages, etc.) are not described in this document, nor are the meanings of Java statements.

The main body of this document describes the language in detail and the appendices provide the full syntax in EBNF and illustrate the language usage with a set of small examples.

The language described here is based largely on the the original Seuss publication notation [Misra 96] and the language *Seuss for C++* [Krüger 96]. Its current form was debated and discussed in the meetings of the Seuss Group at the University of Texas at Austin, attended by Will Adams, Lorenzo Alvisi, Rajeev Joshi, Calvin Lin, Pete Manolios, Jay Misra, Todd Smith and Raymond Tse.

[1] Dept. of Computer Sciences, The University of Texas at Austin, Email: joshi@cs.utexas.edu

Contents

0 Notation and Terminology

EBNF conventions The grammar is described using EBNF, whose conventions and meta symbols are summarised below.

Nonterminals are enclosed by the pair $\langle$ and $\rangle$ (e.g., $\langle$ *program* $\rangle$)
Keywords are in boldface, (e.g., **import**)
Terminal symbols are enclosed within single quotes, (e.g., '-->')
$\{$ E $\}$ represents 0 or more occurrences of E
[E] represents 0 or 1 occurrence of E
(and) are used for grouping
$|$ denotes a choice of productions

As already stated, we are not interested in details of Java syntax. Thus the production rules we describe will contain references to nonterminals which will expand to sections of Java code. We denote all such nonterminals by suffixing them with the word *code* (e.g., $\langle$ *initialisation code* $\rangle$, $\langle$ *body code* $\rangle$). We will also state syntactic and semantic restrictions on the kinds of Java statements and declarations that such a nonterminal may produce. The syntactic restrictions are expected to be checked by the Seuss compiler.

Identifiers and Declarations We use the following terminology to describe the various kinds of identifiers that are needed in the following subsections.

- A *simple identifier* is a string which contains only *Java letters* and *Java digits* [GJS 96], starts with a Java letter and is not a Seuss keyword (appendix B) or Java keyword. Variables b, c will range over simple identifiers.
- A *module identifier* is either a simple identifier or a string of the form $M.c$ where M is a module identifier and c is a simple identifier, e.g., `Library` and `Library.Channels`. Variables K, L, M will range over module identifiers.
- A *qualified identifier* is an identifier of the form $M.c$ where M is a module identifier and c is a simple identifier, e.g., `Library.Channels.Fifo_Channel`. Variables p, q, r will range over qualified identifiers.
- A *general identifier* is either a simple identifier or a qualified identifier. Variables x, y will range over general identifiers.

A simple identifier can be accessed only at those points in a program where it is *visible*. Visibility will be defined in §3.2. At each point in the program where a simple identifier c is visible, it can be replaced by a unique qualified identifier $M.c$, which is called the *resolvent* of c .

A simple identifier b is said to resolve to a *box name* at a program point only if its resolvent is $M.b$ and module M exports a box (§2, §3.1) named b . Similarly, a simple identifier c is said to resolve to an *item name* at a program point only if its resolvent is $M.c$ and M exports an *item* named c .

Restrictions on programs In addition to the syntactic constraints imposed by the grammar, Seuss programs satisfy certain additional restrictions. These restrictions are classified into three categories, based on the kinds of errors that result from violating these restrictions:

- A *static error* can be detected and reported by the compiler, e.g., declaring two variables with the same name.
- A *checked runtime* error can be detected and reported by the runtime system, e.g., accessing an array out of bounds.
- An *unchecked runtime* error may not be detectable by the runtime system, e.g., execution of a function that does not terminate. Neither the compiler nor the runtime system provides any guarantees in the presence of such errors; it is the programmer's responsibility to ensure that they do not arise.

1 Programs and Modules

⟨ *program* ⟩	::=	{ ⟨ *module definition* ⟩ }
⟨ *module definition* ⟩	::=	**module** ⟨ *module identifier* ⟩ '{' ⟨ *module body* ⟩ '}'
⟨ *module identifier* ⟩	::=	⟨ *simple identifier* ⟩ { '.' ⟨ *simple identifier* ⟩ }
⟨ *module body* ⟩	::=	[⟨ *local* ⟩]
		{ ⟨ *entity* ⟩ | ⟨ *import* ⟩ | ⟨ *export* ⟩ | ⟨ *link* ⟩ | ⟨ *needs* ⟩ }
⟨ *local* ⟩	::=	**local** '{' ⟨ *local code* ⟩ '}'

A program is a list of module definitions. A *module* has a *local* section (which is optional) and any combination of *entity* declarations (§2), *import* declarations (§3.0), *export* declarations (§3.1), *link* declarations (§4), and *needs* declarations (§5.0).

The *module identifier* appearing in the rule for *module definition* is called the *name* of that module.

Restriction N (Static) Modules in a program have distinct names.

As a consequence of the restriction above, each module in a program may be referred to unambiguously by its associated name.

Restriction (Static) A program has a module named `main`.

Notation We write "M contains [$decl$]" to denote that the declaration $decl$ occurs in the module named M. The *environment* of a module M in a program consists of the set of modules other than M.

2 Boxes and Items

2.0 Boxes

⟨ *entity* ⟩	::=	⟨ *box* ⟩ | ⟨ *item* ⟩
⟨ *box* ⟩	::=	⟨ *simple box* ⟩ | ⟨ *parameterised box* ⟩
⟨ *simple box* ⟩	::=	**box** ⟨ *simple identifier* ⟩ ⟨ *box body* ⟩

There are two kinds of entities that may be declared within modules – *boxes* and *items*. A box is like a Java class, with variables which define its state and procedures which are used to inspect or change this state. Unlike Java, however, all variables of a box are private to it and can be referred to within the procedures in that box only. Boxes provide a convenient way for declaring commonly used types, e.g., semaphore, channel.

The *simple identifier* appearing in the rule for *simple box* is called the *name* of the box. A box declaration may be parameterised (§2.0.0); however, its name consists only of the simple identifier appearing after the keyword **box** in the declaration.

We say that two identifiers whose resolvent is the same box name denote the *same* type.

2.0.0 Parameterised boxes

⟨ *parameterised box* ⟩	::=	**box** ⟨ *identifier* ⟩ ‘**(**’ ⟨ *box parameter list* ⟩ ‘**)**’ ⟨ *box body* ⟩
⟨ *box parameter list* ⟩	::=	⟨ *box parameter* ⟩ { ‘**,**’ ⟨ *box parameter* ⟩ }
⟨ *box parameter* ⟩	::=	⟨ *identifier* ⟩ ‘**:**’ ⟨ *simple box name* ⟩

Each argument in a parameterised box declaration is of the form `c : X` where `c` is a simple identifier denoting the parameter name and `X` is a simple or qualified identifier denoting a box declared without parameters.

Restriction (Static) The names of the parameters are distinct. The resolvent of the *general identifier* in the rule for *parameterised type* , and the resolvent of the *simple box name* in the rule for *box parameter* , are both box names.

2.1 Items

⟨ *item* ⟩	::=	⟨ *simple item* ⟩ | ⟨ *parameterised item* ⟩
⟨ *simple item* ⟩	::=	**item** ⟨ *identifier list* ⟩ ⟨ *simple item type* ⟩
⟨ *identifier list* ⟩	::=	⟨ *simple identifier* ⟩ { ‘**,**’ ⟨ *simple identifier* ⟩ }
⟨ *simple item type* ⟩	::=	(‘**:**’ ⟨ *simple box name* ⟩ ‘**;**’) | ⟨ *box body* ⟩
⟨ *simple box name* ⟩	::=	⟨ *general identifier* ⟩
⟨ *general identifier* ⟩	::=	⟨ *simple identifier* ⟩ | ⟨ *qualified identifier* ⟩
⟨ *qualified identifier* ⟩	::=	⟨ *module identifier* ⟩ ‘**.**’ ⟨ *simple identifier* ⟩

An item is an instance of a box, thus, it is like a Java object (which is an instance of a Java class).

Convention The compiler replaces every item declaration of the form `item x {` *body* `}` by the two declarations `box Bx {` *body* `}` and `item x : Bx` where `Bx` is a name unique to this declaration which is introduced by the compiler.

With this convention, we define the *type* of an item to be the resolvent of the box name appearing in the item declaration.

Restriction (Static) The entities declared within a module have distinct names. The *simple type name* in an item declaration resolves to a box name.

This restriction, along with **Restriction N** above, allows an entity c declared within module M to be associated with the unique qualified identifier $M.c$.

Example Some examples of simple box and item declarations are given below.

```
module L.M
{      box Fifo {   ...   }
       item ch0 , ch1 : Fifo ;
       item printq { ... }
}
```

These statements declare a module named `L.M` with a box named `Fifo`, two items named `ch0` and `ch1` of type `Fifo` and an item named `printq` of a type with a unique, compiler generated name. The qualified names of the declared entities are `L.M.Fifo, L.M.ch0, L.M.ch1` and `L.M.printq` . Within the module `M`, the box identifiers `Fifo` and `L.M.Fifo` resolve to the same qualified identifier `L.M.Fifo`; thus, they denote the same type.

2.1.0 Parameterised Items

⟨ *parameterised item* ⟩	::=	**item** ⟨ *identifier list* ⟩ ':' ⟨ *parameterised type* ⟩ ';'
⟨ *parameterised type* ⟩	::=	⟨ *general identifier* ⟩ '(' ⟨ *item parameter list* ⟩ ')'
⟨ *item parameter list* ⟩	::=	⟨ *item parameter* ⟩ [',' ⟨ *item parameter* ⟩]
⟨ *item parameter* ⟩	::=	⟨ *general identifier* ⟩

An item of a parameterised box is declared by providing a list of items, one for each parameter.

Restriction (Static) The resolvent of each *item parameter* is an item name. The list of items provided when declaring an item of any box has the same length and sequence of types as the declaration of the item's type.

Example The following example illustrates the syntax for declaring a parameterised box and instantiating it.

```
box B ( chan : Fifo , sem : Semaphore )
{ .. box body, with references to chan and sem .. }

item c : Fifo ;
item s : Semaphore ;
item x : B ( c , s ) ;
```

2.2 Box Body

⟨ *box body* ⟩	::=	'{' ⟨ *local declarations* ⟩ ⟨ *procedures* ⟩ '}'
⟨ *local declarations* ⟩	::=	[⟨ *box locals* ⟩] [⟨ *initialisations* ⟩]

$\langle$ *box locals* $\rangle$ ::= **local** '{' $\langle$ *local code* $\rangle$ '}'
$\langle$ *initialisations* $\rangle$ ::= **init** '{' $\langle$ *initialisation code* $\rangle$ '}'

The body of a box consists of a local section and procedure declarations. The *local declarations* define the types, variables and functions that are used by the procedures declared later; they may also specify *box initialistations* which consist of a Java program which is executed when an item is created at the beginning of an execution.

Restriction (Static) The undeclared identifiers in *local code* refer to Java types declared within the local declarations for the current module. The undeclared identifiers in *initialisation code* either (i) are declared in the local section for this box, (ii) are declared in the local section of the current module, or (iii) have resolvents that are item names.

Restriction (Checked runtime) The *initialisation code* and the procedures in a box may raise exceptions; it is up to the implementation to decide whether to abort the program execution.

Restriction (Unchecked runtime) The *initialisation code* is a terminating Java program.

Examples An example of local declarations is shown below.

```
box Semaphore
{
    local
    {   class Queue { .. }
        Queue q ;  boolean avail ;
        boolean ready(int t) { ... }
    }
    init {  avail = true ; }

    ...
}
```

The first line within the local section defines a new Java class called `Queue` which is to be used to store a sequence of integers. The second line defines two variables – an instance of the Queue class and a boolean variable for denoting whether the semaphore is available. The third line defines a locally accessible function `ready`, which has return type boolean and one value parameter of integer type. The fourth line defines the initialisation commands for the box `Semaphore` : the variable `avail` is to be initialised to `true`.

Procedures

$\langle$ *procedures* $\rangle$ ::= { $\langle$ *method* $\rangle$ | $\langle$ *action* $\rangle$ }

A procedure may be a method or an action; these may occur in any order in the box body. Methods may be invoked by procedures in other items, thus they are named and may take parameters. Actions are autonomously executing procedures which cannot be invoked by other procedures; thus they may be unnamed and do not take any parameters.

Notation We say that a method is declared for an item if it is declared in the box for that item.

Restriction (Static) No two named procedures in a box have the same name.

2.3 Methods

There are two kinds of methods, *total methods* and *partial methods*. A total method is a nonblocking program which is expected to terminate; its syntax is very similar to that of class member functions in Java. A partial method has a special form, which is described below(§2.3.1).

The syntax for a method call is the same as the syntax for a member function call in Java. For instance, `ch.put(n)` denotes a call with the parameter `n` to the method `put` in item `ch`.

Notation For a method f in an item with resolvent q that contains a call to a method g in an item with resolvent r, we write $q.f$ *calls* $r.g$.

Requirement Acyclicity (Static) The transitive closure of the *calls* relation defined above is acyclic.

2.3.0 Total Methods

⟨ *method* ⟩	::=	⟨ *total method* ⟩ | ⟨ *partial method* ⟩
⟨ *total method* ⟩	::=	**total method** ⟨ *total method head* ⟩ ⟨ *total command* ⟩
⟨ *total method head* ⟩	::=	⟨ *type specifier* ⟩ ⟨ *procedure head* ⟩
⟨ *type specifier* ⟩	::=	⟨ *general identifier* ⟩
⟨ *procedure head* ⟩	::=	⟨ *simple identifier* ⟩ ‘(’ [⟨ *formal parameters* ⟩] ‘)’
⟨ *formal parameters* ⟩	::=	⟨ *argument declaration* ⟩ { ‘,’ ⟨ *argument declaration* ⟩ }
⟨ *argument declaration* ⟩	::=	⟨ *type specifier* ⟩ ⟨ *simple identifier* ⟩
⟨ *total command* ⟩	::=	‘{’ ⟨ *body code* ⟩ ‘}’

The *type specifier* in *total method head* defines the return type of the total method. As in Java, a type specifier of `void` declares a pure procedure. The body can contain any Java statements and declarations.

Restriction (Static) All *type specifiers* occurring in *total method head* and *argument declaration* either (i) are built in Java types, (ii) are declared in the local code for the box, or (iii) are declared in the local code for the current module.

Restriction (Static) The undeclared identifiers in *body code* either (i) are declared in the local section of the box, (ii) are declared in the local section of the current module, or (ii) resolve to item names.

Restriction (Static) The body code does not contain any calls to partial methods.

Restriction (Unchecked runtime) The body is a terminating Java program.

2.3.1 Partial Methods

⟨ *partial method* ⟩	::=	**partial method** ⟨ *procedure head* ⟩ ⟨ *partial command* ⟩
⟨ *partial command* ⟩	::=	'{' [⟨ *local* ⟩] ⟨ *alternative list* ⟩ '}'
⟨ *alternative list* ⟩	::=	(⟨ *alternative* ⟩ { ⟨ *alternative type* ⟩ ⟨ *alternative* ⟩ })
⟨ *alternative type* ⟩	::=	'`[+]`' | '`[-]`'
⟨ *alternative* ⟩	::=	(⟨ *precondition* ⟩ ['**;**' ⟨ *preprocedure* ⟩] '`-->`' ⟨ *total command* ⟩)
	|	('**;**' ⟨ *preprocedure* ⟩ '`-->`' ⟨ *total command* ⟩)
⟨ *precondition* ⟩	::=	'(' ⟨ *boolean code* ⟩ ')'
⟨ *preprocedure* ⟩	::=	⟨ *qualified method name* ⟩ '(' ⟨ *parameter code* ⟩ ')'
⟨ *qualified method name* ⟩	::=	⟨ *item name* ⟩ '**.**' ⟨ *simple identifier* ⟩
⟨ *item name* ⟩	::=	⟨ *general identifier* ⟩

A partial method is a pure procedure, with no return type mentioned in its header. It consists of local Java declarations followed by a nonempty list of *alternatives*. Variables and functions declared as locals are visible only within the alternatives. An alternative may be *positive* or *negative*, as indicated by the prefix '`[+]`' or '`[-]`'. The first alternative has no prefix; it is considered to be a positive alternative. The *precondition* is a Java expression; it is expected to be of boolean type. The *preprocedure* is a call to a partial method of another item. Either the *precondition* or the *preprocedure* may be absent, but not both. When the precondition is omitted, it is assumed to be equivalent to `true`.

A partial method *accepts* or *rejects* each call made upon it. When a partial method is called, the preconditions of its alternatives are evaluated (in unspecified order) and the unique alternative whose precondition is true (see restriction below) is invoked. The method accepts if and only if some positive alternative accepts the call; the method rejects the call otherwise. An alternative of the form `p --> S`, with precondition `p` and total command `S`, always accepts calls made upon it; its execution consists of executing `S` . An alternative of the form `p ; f --> S`, with precondition `p`, preprocedure `f` and total command `S`, executes by first calling `f`. If the call on `f` is accepted, then `S` is executed, and the alternative accepts, otherwise, if the call on `f` is rejected, the alternative rejects.

Restrictions The arguments and total command in a partial method satisfy the same restrictions as those described for total methods.

Restriction (Static) The undeclared identifiers in *boolean code* and *parameter code* are declared in the local section of the current procedure, (ii) the local section of the current box, or (iii) the current section of the current module. The *preprocedure* is a valid partial method name for the *item name* . It has a signature [GJS 96] that matches *parameter code*.

Restriction (Unchecked runtime) The preconditions of the alternatives in a partial method are disjoint. A precondition may cause side-effects only from states in which it evaluates to true and only when the preprocedure is absent.

2.4 Actions

⟨ *action* ⟩	::=	⟨ *simple action* ⟩ | ⟨ *quantified action* ⟩

$$\begin{array}{rcl}
\langle \textit{simple action} \rangle & ::= & \langle \textit{simple total action} \rangle \quad | \quad \langle \textit{simple partial action} \rangle \\
\langle \textit{simple total action} \rangle & ::= & \textbf{total action} \ \ [\ \langle \textit{simple identifier} \rangle\ ] \ \ \langle \textit{total command} \rangle \\
\langle \textit{simple partial action} \rangle & ::= & \textbf{partial action} \ \ [\ \langle \textit{simple identifier} \rangle\ ] \ \ \langle \textit{partial command} \rangle
\end{array}$$

As stated above, an action is a procedure that cannot be invoked by other procedures; instead, the scheduler guarantees that every action is executed infinitely often in an execution. Quantified actions define collections of parameterised actions; they are described below. The restrictions stated for total (partial) methods also apply to total (partial) actions.

Restriction (Static) A partial action does not have any negative alternatives.

Quantified Actions

$$\begin{array}{rcl}
\langle \textit{quantified action} \rangle & ::= & \textbf{forall} \ \ \langle \textit{identifier} \rangle \ \ \text{‘}\mathbf{:}\text{’} \ \ \langle \textit{range} \rangle \ \ \langle \textit{quantified body} \rangle \\
\langle \textit{range} \rangle & ::= & \langle \textit{constant} \rangle \ \ \text{‘}\mathbf{..}\text{’} \ \ \langle \textit{constant} \rangle \\
\langle \textit{quantified body} \rangle & ::= & \langle \textit{action} \rangle \quad | \quad (\ \text{‘}\{\text{’} \ \ \langle \textit{actions} \rangle \ \ \text{‘}\}\text{’}\ ) \\
\langle \textit{actions} \rangle & ::= & \langle \textit{action} \rangle \ \ \{\ \langle \textit{action} \rangle\ \}
\end{array}$$

A quantified action is a collection of actions parameterised by an index variable, which may appear in the body of the action. The range of this variable is required to be determinable at compilation time, hence it is restricted to be of the form $l \dots h$ where l and h are integer constants.

When $l \leq h$, such a quantified action declaration with index variable j is equivalent to writing $h-l+1$ action declarations, one for each value n in the range $l \dots h$, with free occurrences of j in *quantified body* being replaced by n.

3 Imports and Exports

Import declarations allow identifiers declared in one module to be used in another without qualification. Export declarations identify which identifiers are to be made visible outside a module; they serve the same purpose as *public* declarations in Java packages.

3.0 Imports

$$\begin{array}{rcl}
\langle \textit{import} \rangle & ::= & \langle \textit{module import} \rangle \quad | \quad \langle \textit{entity import} \rangle \\
\langle \textit{module import} \rangle & ::= & \textbf{import} \ \ \langle \textit{module name list} \rangle \ \ \text{‘}\mathbf{;}\text{’} \\
\langle \textit{module name list} \rangle & ::= & \langle \textit{module name} \rangle \ \ \{\ \text{‘}\mathbf{,}\text{’}\ \langle \textit{module name} \rangle\ \} \\
\langle \textit{entity import} \rangle & ::= & \textbf{from} \ \ \langle \textit{module name} \rangle \ \ \textbf{import} \ \ \langle \textit{identifier list} \rangle \ \ \text{‘}\mathbf{;}\text{’}
\end{array}$$

As in Java, an import declaration allows identifiers exported by another module to be referred to by a simple name consisting of a single identifier. In the absence of an import declaration, an entity exported by a module can be referenced in another module only by using a qualified identifier. There are two kinds of import declarations – *module imports* and *entity imports*. A module import has the form `import` *list* where *list* is a list of module identifiers; such a declaration is equivalent to writing `import` M for each M in *list*. We say that the declaration `import` M *implicitly imports* c from M for each simple identifier

c exported by M (§3.1). An entity import has the form `from` M `import` *list* where *list* is a list of simple identifier; such a declaration has the same effect as writing `from` M `import` c for each c in *list*. We say that the declaration `from` M `import` c *explicitly imports* c from M. We write *imported* to mean explicitly imported or implicitly imported.

Restriction (Static) A simple identifier may be imported from a module M only if it has been exported by M.

Examples An example of the use of imports is shown below. Assume that the box `Fifo` is declared in the module `Buffers` and that `WeakSemaphore` and `StrongSemaphore` are declared in the module `Semaphores`.

```
module User
{     from Semaphores import WeakSemaphore ;
      import Buffers ;

      item ch : Fifo ;
      item wsem : WeakSemaphore ;
      item ssem : Semaphores.StrongSemaphore ;
}
```

3.1 Exports

$$\begin{array}{rcl} \langle \textit{export} \rangle & ::= & \textbf{export} \ \langle \textit{unit name list} \rangle \ \text{`;'} \\ \langle \textit{unit name list} \rangle & ::= & \langle \textit{unit name} \rangle \ \{ \ \text{`,'} \ \langle \textit{unit name} \rangle \ \} \\ \langle \textit{unit name} \rangle & ::= & \langle \textit{simple identifier} \rangle \ [\ \text{`('} \ \langle \textit{general identifier list} \rangle \ \text{`)'} \ ] \end{array}$$

Export declarations allow entities declared in a module to be imported (§3.0) by other modules. An export declaration has the form `export` *list* where *list* is a list of entity names; such a declaration has the same effect as writing `export` x for every x in *list*. We say that the declaration `export` x results in the identifier x being *exported by* the current module.

Restriction (Static) Every exported identifier resolves to an entity name. If a parameterised box is exported, then each simple identifier occuring in the parameter list is explicitly exported by the module.

Example As an illustration, we note that the following example violates the restriction above:

```
module Incorrect
{     from Buffers import Fifo ;

      box B ( chan : Fifo ) { ... }

      export B ( Fifo ) ;
}
```

The example would be legal if the declaration `export Fifo` precedes `export B ( Fifo )`, or if the latter were changed to `export B ( Buffers.Fifo )`.

3.2 Names

We say that a module K is *below* a module M, denoted $K \sqsubset M$, provided any of the following holds:

- M contains [`import` K]
- for some x, M contains [`from` K `import` x]
- for some module L, $K \sqsubset L$ and $L \sqsubset M$

Restriction (Static) The relation $\sqsubset$ over modules is asymmetric.

Restriction U (Static) For any module M, the collection of simple identifiers consisting of

- the identifiers declared in M
- the identifiers explicitly imported from other modules
- the identifiers implicitly imported from other modules

does not contain any duplicates.

The collection described above therefore corresponds to a *set* of identifiers; we refer to it as the set of *visible* identifiers of M.

Resolution Using **Restriction U**, a simple identifier c that is visible in M corresponds to a unique qualified identifier $K.c$, as follows. If c is declared in M then K is M; otherwise, c is imported and K is the module from which it is imported. We say that $K.c$ is the *resolvent* of c and that c *resolves to* $K.c$.

Notation A qualified identifier resolves to itself.

4 Link Declarations

⟨ *link* ⟩	::=	⟨ *inner* ⟩ | ⟨ *outer* ⟩
⟨ *inner* ⟩	::=	**inner** ⟨ *item name* ⟩ ⟨ *method name list* ⟩ ‘**;**’
⟨ *outer* ⟩	::=	**outer** ⟨ *item name* ⟩ ⟨ *method name list* ⟩ ‘**;**’
⟨ *method name list* ⟩	::=	(‘+’ | ‘−’) ‘{’ ⟨ *method name* ⟩ { ‘**,**’ ⟨ *method name* ⟩ } ‘}’
⟨ *method name* ⟩	::=	⟨ *identifier* ⟩

Link declarations are used to state assumptions about the environment and to assert guarantees for it. There are two kinds of link declarations; *inner* declarations and *outer* declarations.

Notation Let S denote the set of methods of an item. For any subset T of S, we write $\overline{T}$ to mean the set $S \setminus T$.

A declaration of the form `inner x + {` T `}` asserts that items declared in the current module may invoke only those methods of `x` that are present in T. A declaration of the form `inner x - {` T `}` asserts that items declared in the current module may invoke only those methods of `x` that are present in $\overline{T}$.

A declaration of the form `outer x + {` T `}` states that items in the environment (§1) of the current module may invoke only those methods of `x` that are present in T. A declaration of the form `outer x - {` T `}` states that items in the environment of the current module may invoke only those methods of `x` that are present in $\overline{T}$.

Restriction (Static) A module contains at most one outer and one inner declaration for any item. Each identifier in *method name list* is the name of a method declared in that item.

Notation Let q denote the resolvent of identifier x in module M. Then,

- If M contains [`inner` x `+` T], we write $(q, T) \in M.inner$
- If M contains [`inner` x `-` T], we write $(q, \overline{T}) \in M.inner$

We use a similar convention for outer declarations.

Restriction (Static) If $(q, T) \in M.inner$, an item declared within module M can invoke only those methods on q that are in T.

Restriction (Static) The inner and outer declarations in a program satisfy the following property, which is called the *Link Constraint* [Misra 94].

$$(\forall\, M, K, q, S, T \;::\; (M \neq K \;\wedge\; (q, S) \in M.inner \;\wedge\; (q, T) \in K.outer) \;\Rightarrow\; S \subseteq T)$$

5 Program Execution

In this section, we explain the syntax for *needs* declarations and discuss how they are used to define the executions of a Seuss program.

5.0 Needs

$\langle$ *needs* $\rangle$ ::= $\langle$ *simple identifier* $\rangle$ **needs** $\langle$ *item name list* $\rangle$ **';'**
$\langle$ *item name list* $\rangle$ ::= $\langle$ *item name* $\rangle$ { **','** $\langle$ *item name* $\rangle$ }

`needs` declarations provide a way for associating a list of items with an entity which are used by the linker to create the items required at runtime. For an entity x, the declaration x `needs` *`ilist`* is equivalent to writing x `needs` y for each item y in *ilist*. For items x, y with resolvent q, r respectively, the declaration x `needs` y means that in any execution with run set R (§5.1), $q \in R \Rightarrow r \in R$. For a box b and item y with resolvents q, r respectivley, the declaration b `needs` y means that in any execution with run set R, if R contains an item whose type resolves to q, then $r \in R$.

Restriction (Static) The *simple identifier* in the rule for *needs* resolves to an entity declared in the current module. Every item in *item name list* resolves to an item name.

We define the relation $needs$ between items as the smallest relation on qualified identifiers satisfying the following conditions. For all M, c, p, q, r, x,

- if M contains [c `needs` x] and x resolves to q, then $M.c$ $needs$ q
- if p is an item of box q and q $needs$ r, then p $needs$ r
- if there is some q such that p $needs$ q and q $needs$ r, then p $needs$ r

5.1 Run Set

Let Q denote the set of simple identifiers visible in the main module that resolve to item names. The *run set* R for a Seuss program is the smallest superset of Q that is closed under the $needs$ relation.

Executing a program with run set R proceeds by first executing the initialisation sections (§2.2) of the items in R subject to the following restriction: for any modules M, K such that $K \sqsubset M$ (§3.0), the items in K are initialised before the items in M.

Thereafter, execution consists of repeatedly choosing an arbitrary action from an item in R and executing it, subject to the constraint that every action be executed infinitely often.

Example

```
module M
{     item sem { ... } ;
      item lw9 { .. accesses sem .. }

      box Fifo { ... }
      box Printer ( chan : Fifo ) { .. accesses chan, sem .. }

      lw9 needs sem ;
      Printer needs sem ;

      export lw9 ;
      export Printer ( Library.Fifo ) ;
}

module main
{     item ch : M.Fifo ;
      item pr : M.Printer ( ch ) ;
}
```

Now executions of this program will include the items `main.ch, main.pr` and `M.sem` but not the item `M.lw9`.

6 Pragmas and Comments

$\langle$ *pragma* $\rangle$ ::= **pragma** '{' $\langle$ *pragma string* $\rangle$ '}'

Pragmas are Seuss-specific constructs, which are meant to provide hints to implementations. They may occur anywhere in a Seuss program, except within comments, strings and Java code sections. It is expected that ignoring pragma declarations will not change the semantics of a program. The current language definition does not require any particular pragmas to be supported.

The comments in a Seuss program are identical to Java comments; they may occur anywhere within the program.

A EBNF

⟨ *program* ⟩ ::= { ⟨ *module definition* ⟩ }

⟨ *module definition* ⟩ ::= **module** ⟨ *module identifier* ⟩ ‘{’ ⟨ *module body* ⟩ ‘}’

⟨ *module identifier* ⟩ ::= ⟨ *simple identifier* ⟩ { ‘**.**’ ⟨ *simple identifier* ⟩ }

⟨ *module body* ⟩ ::= [⟨ *local* ⟩]
{ ⟨ *entity* ⟩ | ⟨ *import* ⟩ | ⟨ *export* ⟩ | ⟨ *link* ⟩ | ⟨ *needs* ⟩ }

⟨ *local* ⟩ ::= **local** ‘{’ ⟨ *local code* ⟩ ‘}’

⟨ *entity* ⟩ ::= ⟨ *box* ⟩ | ⟨ *item* ⟩

⟨ *box* ⟩ ::= ⟨ *simple box* ⟩ | ⟨ *parameterised box* ⟩

⟨ *simple box* ⟩ ::= **box** ⟨ *simple identifier* ⟩ ⟨ *box body* ⟩

⟨ *parameterised box* ⟩ ::= **box** ⟨ *identifier* ⟩ ‘(’ ⟨ *box parameter list* ⟩ ‘)’ ⟨ *box body* ⟩

⟨ *box parameter list* ⟩ ::= ⟨ *box parameter* ⟩ { ‘**,**’ ⟨ *box parameter* ⟩ }

⟨ *box parameter* ⟩ ::= ⟨ *identifier* ⟩ ‘**:**’ ⟨ *simple box name* ⟩

⟨ *item* ⟩ ::= ⟨ *simple item* ⟩ | ⟨ *parameterised item* ⟩

⟨ *simple item* ⟩ ::= **item** ⟨ *identifier list* ⟩ ⟨ *simple item type* ⟩

⟨ *identifier list* ⟩ ::= ⟨ *simple identifier* ⟩ { ‘**,**’ ⟨ *simple identifier* ⟩ }

⟨ *simple item type* ⟩ ::= (‘**:**’ ⟨ *simple box name* ⟩ ‘**;**’) | ⟨ *box body* ⟩

⟨ *simple box name* ⟩ ::= ⟨ *general identifier* ⟩

⟨ *general identifier* ⟩ ::= ⟨ *simple identifier* ⟩ | ⟨ *qualified identifier* ⟩

⟨ *qualified identifier* ⟩ ::= ⟨ *module identifier* ⟩ ‘**.**’ ⟨ *simple identifier* ⟩

⟨ *parameterised item* ⟩ ::= **item** ⟨ *identifier list* ⟩ ‘**:**’ ⟨ *parameterised type* ⟩ ‘**;**’

⟨ *parameterised type* ⟩ ::= ⟨ *general identifier* ⟩ ‘**(**’ ⟨ *item parameter list* ⟩ ‘**)**’

⟨ *item parameter list* ⟩ ::= ⟨ *item parameter* ⟩ [‘**,**’ ⟨ *item parameter* ⟩]

⟨ *item parameter* ⟩ ::= ⟨ *general identifier* ⟩

⟨ *box body* ⟩ ::= ‘{’ ⟨ *local declarations* ⟩ ⟨ *procedures* ⟩ ‘}’

⟨ *local declarations* ⟩ ::= [⟨ *box locals* ⟩] [⟨ *initialisations* ⟩]

⟨ *box locals* ⟩ ::= **local** ‘{’ ⟨ *local code* ⟩ ‘}’

⟨ *initialisations* ⟩ ::= **init** ‘{’ ⟨ *initialisation code* ⟩ ‘}’

⟨ *procedures* ⟩ ::= { ⟨ *method* ⟩ | ⟨ *action* ⟩ }

⟨ *method* ⟩ ::= ⟨ *total method* ⟩ | ⟨ *partial method* ⟩

⟨ *total method* ⟩ ::= **total method** ⟨ *total method head* ⟩ ⟨ *total command* ⟩

⟨ *total method head* ⟩ ::= ⟨ *type specifier* ⟩ ⟨ *procedure head* ⟩

⟨ *type specifier* ⟩ ::= ⟨ *general identifier* ⟩

⟨ *procedure head* ⟩ ::= ⟨ *simple identifier* ⟩ ‘(’ [⟨ *formal parameters* ⟩] ‘)’

⟨ *formal parameters* ⟩ ::= ⟨ *argument declaration* ⟩ { ‘**,**’ ⟨ *argument declaration* ⟩ }

⟨ *argument declaration* ⟩ ::= ⟨ *type specifier* ⟩ ⟨ *simple identifier* ⟩

⟨ *total command* ⟩ ::= ‘{’ ⟨ *body code* ⟩ ‘}’

⟨ *partial method* ⟩ ::= **partial method** ⟨ *procedure head* ⟩ ⟨ *partial command* ⟩

⟨ *partial command* ⟩ ::= ‘{’ [⟨ *local* ⟩] ⟨ *alternative list* ⟩ ‘}’

⟨ *alternative list* ⟩ ::= (⟨ *alternative* ⟩ { ⟨ *alternative type* ⟩ ⟨ *alternative* ⟩ })
⟨ *alternative type* ⟩ ::= ‘`[+]`’ | ‘`[-]`’
⟨ *alternative* ⟩ ::= (⟨ *precondition* ⟩ [‘**;**’ ⟨ *preprocedure* ⟩] ‘`-->`’ ⟨ *total command* ⟩)
| (‘**;**’ ⟨ *preprocedure* ⟩ ‘`-->`’ ⟨ *total command* ⟩)
⟨ *precondition* ⟩ ::= ‘**(**’ ⟨ *boolean code* ⟩ ‘**)**’
⟨ *preprocedure* ⟩ ::= ⟨ *qualified method name* ⟩ ‘**(**’ ⟨ *parameter code* ⟩ ‘**)**’
⟨ *qualified method name* ⟩ ::= ⟨ *item name* ⟩ ‘**.**’ ⟨ *simple identifier* ⟩
⟨ *item name* ⟩ ::= ⟨ *general identifier* ⟩
⟨ *action* ⟩ ::= ⟨ *simple action* ⟩ | ⟨ *quantified action* ⟩
⟨ *simple action* ⟩ ::= ⟨ *simple total action* ⟩ | ⟨ *simple partial action* ⟩
⟨ *simple total action* ⟩ ::= **total action** [⟨ *simple identifier* ⟩] ⟨ *total command* ⟩
⟨ *simple partial action* ⟩ ::= **partial action** [⟨ *simple identifier* ⟩] ⟨ *partial command* ⟩
⟨ *quantified action* ⟩ ::= **forall** ⟨ *identifier* ⟩ ‘**:**’ ⟨ *range* ⟩ ⟨ *quantified body* ⟩
⟨ *range* ⟩ ::= ⟨ *constant* ⟩ ‘**..**’ ⟨ *constant* ⟩
⟨ *quantified body* ⟩ ::= ⟨ *action* ⟩ | (‘{’ ⟨ *actions* ⟩ ‘}’)
⟨ *actions* ⟩ ::= ⟨ *action* ⟩ { ⟨ *action* ⟩ }
⟨ *import* ⟩ ::= ⟨ *module import* ⟩ | ⟨ *entity import* ⟩
⟨ *module import* ⟩ ::= **import** ⟨ *module name list* ⟩ ‘**;**’
⟨ *module name list* ⟩ ::= ⟨ *module name* ⟩ { ‘**,**’ ⟨ *module name* ⟩ }
⟨ *entity import* ⟩ ::= **from** ⟨ *module name* ⟩ **import** ⟨ *identifier list* ⟩ ‘**;**’
⟨ *export* ⟩ ::= **export** ⟨ *unit name list* ⟩ ‘**;**’
⟨ *unit name list* ⟩ ::= ⟨ *unit name* ⟩ { ‘**,**’ ⟨ *unit name* ⟩ }
⟨ *unit name* ⟩ ::= ⟨ *simple identifier* ⟩ [‘**(**’ ⟨ *general identifier list* ⟩ ‘**)**’]
⟨ *link* ⟩ ::= ⟨ *inner* ⟩ | ⟨ *outer* ⟩
⟨ *inner* ⟩ ::= **inner** ⟨ *item name* ⟩ ⟨ *method name list* ⟩ ‘**;**’
⟨ *outer* ⟩ ::= **outer** ⟨ *item name* ⟩ ⟨ *method name list* ⟩ ‘**;**’
⟨ *method name list* ⟩ ::= (‘+’ | ‘−’) ‘{’ ⟨ *method name* ⟩ { ‘**,**’ ⟨ *method name* ⟩ } ‘}’
⟨ *method name* ⟩ ::= ⟨ *identifier* ⟩
⟨ *needs* ⟩ ::= ⟨ *simple identifier* ⟩ **needs** ⟨ *item name list* ⟩ ‘**;**’
⟨ *item name list* ⟩ ::= ⟨ *item name* ⟩ { ‘**,**’ ⟨ *item name* ⟩ }
⟨ *pragma* ⟩ ::= **pragma** ‘{’ ⟨ *pragma string* ⟩ ‘}’

B Keywords

```
action   box      export   forall   from     import   init
inner    item     local    main     method   module   needs
outer    partial  pragma   total
```

C Examples

Ex. 0: Total methods.
The item `counter0` below implements a counter which takes on non-negative values. The item has two methods. Method **inc** increments the counter, and method **fetch** returns the current value; it also resets the counter to 0.

```
item counter0
{
    local { int n; }
    init { n = 0; }

    total method void inc()  { n = n+1; }

    total method int fetch()
    {
        int t = n;
        n = 0;
        return t;
    }
} // item counter0
```

Ex. 1: Partial method with a positive alternative.
The box below defines an unbounded integer Fifo buffer with a total method `put`, and a partial method `get`. It uses the `Vector` class which is available in the standard Java libraries.

```
module Channels
{
    local { import java.util.Vector ;
            class Nat { public int val ; }
          }

    box intFifo
    {
        local { Vector q ; }

        init { // Create a new vector with 16 elements
               // Vector should be doubled as needed (2nd argument)

               q = new Vector(16, 0) ;
             }

        total method void put(Nat n)
        {    q.addElement(n) ;    }
```

```
        partial method get(Nat n)
        {   (!q.isEmpty()) --> { n = q.elementAt(0) ; q.removeElementAt(0) ;
                                 if (2*q.size() < q.capacity())
                                    q.trimToSize() ;
                               }
        }
    } // box intFifo

} // module Channels
```

<u>Ex. 2:</u> Partial method with negative alternatives.
In the following definition of a strong binary sempahore, we define a type `Ticket` which has a method `new_value` which returns a new, non-`Nil` natural number on each call.

```
module Semaphore.Binary
{   local
    {   import java.util.Vector ;

        class Nat { public int val ; }

        class Ticket
        {   static int x = 0 ;

            public Nat new_value() { Nat z = new Nat() ; z.val = x ;
                                       ++x ; return z ; }
        }
    }

    box StrongBinarySemaphore
    {
        local
        {       Ticket tk ;
                boolean avail ;
                Vector q ;
        }

        init { avail = true ; q = new Vector(16,0) ; }

        partial method P(Nat n)
        {      ((n != null) && avail && (q.elementAt(0) == n)) -->
                                  { n = null ;
                                    q.removeElementAt(0) ;
                                    avail = false ; }

          [-]  (n == null) -->    { n = tk.new_value() ;
```

```
                                    q.addElement(n) ; }
        }

        total method void V()   {   avail = true ;   }

    } // box StrongBinarySemaphore
} // module Semaphore.Binary
```

EX. 3: Using local procedures.
Consider a generalisation of the above to implement a strong general semaphore. A simple solution is to do the following: replace the `boolean` variable `avail` by a variable of type integer, initialised to 0, replace the assignment in `V` by an increment operation and the assignment in `P` by a decrement operation. However, this solution has the undesirable property that, even when the semaphore value is greater than 1, a process waiting for the semaphore must wait until it is at the head of the queue. A better solution is to allow a process to acquire the semaphore if it is among the first k entries in the queue, where k is the current value of the semaphore. This solution is implemented below. The variable `avail`, now of type `Nat`, holds the value of the semaphore. The local function `ready` works as follows. If the Ticket parameter `n` is among the first `avail` entries in the queue, the function returns `true`; otherwise, it returns `false`.

```
module Semaphore.General
{   local
    {   import java.util.Vector ;

        class Nat { public int val ; }

        class Ticket
        {   static int x = 0 ;

            public Nat new_value() { Nat z = new Nat() ; z.val = x ;
                                        ++x ; return z ; }
        }
    }

    box StrongGeneralSemaphore
    {
        local
        {   Vector q ; Ticket tk ;
            unsigned int avail ;
            const unsigned int K = 5 ;  // Initial value of the semaphore

            boolean ready(Nat n)
            {    // Test whether n is among the first avail entries in q.

                 return ((q.indexOf(n) != -1) && q.indexOf(n) < avail) ;
            }
```

```
        }

        init { avail = K ; tk = new Ticket() ; }

        partial method P(Nat n)
        {
               ((n != null) && ready(n)  -->
                                { n = null ;
                                  q.removeElement(n) ;
                                  --avail ; }

          [-]  (n == null) --> { n = tk.new_value() ;
                                  q = q.addElement(n) ; }
        }

        total method void V()  {  ++avail ;  }

    } // box StrongGeneralSemaphore
```

Ex. 4: (Preprocedures in positive alternatives.)
The item `interleaver` shown below has a single partial method `next`, which alternately returns elements from two input channels, called `in0` and `in1`.

```
module M
{   from Channels import Fifo ;

    item ch0, ch1 : Fifo ;

    item interleaver
    {
        local { Boolean b ; }

        init { b = False ; }

        partial method next(Nat n)
        {
                (!b) ; in0.get(n)  -->  { b = True ;  }
           [+]  ( b) ; in1.get(n)  -->  { b = False ; }
        }

    } // item interleaver
} // M
```

Ex. 5: (Preprocedures in negative alternatives.)
Consider this simple variation of a classic problem in resource allocation. A set of processes are sharing three

resources, which are to be used exclusively. Each process works as follows : for each resource that it needs, it first attempts to acquire an associated lock (encoded as a semaphore); when all needed locks have been acquired, it uses the resources and then releases them.

If different processes attempt to acquire the semaphores in different orders, they may end up in a deadly embrace. One way to avoid this is to ensure that all processes attempt to acquire the resources in the same order. This can be done by defining a shared box which acquires the semaphores on behalf of the processes; the box is designed so that it attempts to acquire semaphores in the same order for all each process.

In the solution below, the item `collector` has a partial method `procure`, which acquires the semaphores on behalf of a waiting processes, and a total method `release`, which releases the acquired semaphores. The procurement of semaphores is in the same order for each process; thus deadlock is avoided. The variable `t` passed to `procure` is used to determine which semaphores the calling process still needs.

```
module M
{   import Semaphore.Binary ;

    item sem0, sem1, sem2 : StrongBinarySemaphore ;

    item collector
    {
        partial method procure(Natural t, boolean needs[])
        {
                 (t.val > 2)  -->  { t = 0; }
            [-]  ((t.val == 2) && needs[2]) ; sem2.P()   --> { ++t ; }
            [-]  ((t.val == 1) && needs[1]) ; sem1.P()   --> { ++t ; }
            [-]  ((t.val == 0) && needs[0]) ; sem0.P()   --> { ++t ; }
            [-]  ((t.val <= 2) && !needs[t.val])         --> { ++t ; }
        }

        total method release(BooleanArray needs)
        {
            if (needs[0])  sem0.V() ;
            if (needs[1])  sem1.V() ;
            if (needs[2])  sem2.V() ;
        }
    } // item collector
} // module M
```

An interesting feature of this solution is the use of preprocedures in negative alternatives. In particular, method `procure` rejects after every `P` attempt, regardless of whether the attempt returned with an acceptance or a rejection.

<u>Ex. 6:</u> Simple example with actions.
This example illustrates how to implement a simple arbitrary natural number generator using actions. The item `anat` has two actions, labelled `inc` and `dec`, which increment and decrement the counter respectively.

```
    item anat
    {
```

```
        local { Nat n; }

        init { n.val = 0; }

        total action inc { n.val = n.val + 1 ; }

        partial action dec { (n.val > 0) --> { n.val = n.val - 1 ; } }■

        total method Nat fetch()
        {
            Nat t = new Nat() ;
            t.val = n.val ;
            n.val = 0 ;
            return t ;
        }
    } /* item anat */
```

Note that it is possible for `fetch` to return 0 on each call. One way to guarantee that a positive number is always returned eventually is to remove the action labelled `dec`.

Ex. 7: Module definitions and imports.

```
module Library.Channels
{
     box Fifo
     {  ... }

     box BoundedBuffer
     {  ... }

     export Fifo, BoundedBuffer ;
} // module Library.Channels

module Semaphores.Weak
{
     box GeneralSemaphore
     {  ... }

     box BinarySemaphore
     {  ... }

     export GeneralSemaphore, BinarySemaphore ;
} // module Semaphores.Weak

module Semaphores.Strong
```

```
{
     box GeneralSemaphore
     {  ... }

     box BinarySemaphore
     {  ... }

     export GeneralSemaphore, BinarySemaphore ;
} // module Semaphores.Strong

module IO
{
     import Libary.Channel ;

     item job_queue : Fifo ;

     box Printer { ... uses job_queue ... }

     export Printer ;
} // module IO

module Library.Devices
{
     import IO ;

     item lw9 : Printer ;
} // module Library.Devices
```

Ex. 8: Item imports.
The following box implements a program that locks a terminal each time the total method `lock` is invoked. No more interaction is possible until the correct password is entered at the keyboard. Items `Keyboard` and `Display` are assumed to be available in Library.Devices, as shown. In the item `ScreenSaver`, the access constraints specify that no other item may access the keyboard, although the display may be shared. The variable `Password` is assumed to be a constant valued string containing the password that unlocks the terminal.

```
module Library.Devices
{
    item Keyboard { .. body with method getword .. }
    item Display { .. body with method put.. }

    export Keyboard , Display ;

} // module Library.Devices
```

```
module SS
{
    import Devices ;

    inner Keyboard + { getword }
    outer Keyboard - { getword }

    inner Display  + { put }

    item ScreenSaver
    {
        local { boolean b;  String w, pword ; }

        init { b = True ; }

        total method setPassword(String p) { pword = new String(p) ; }

        total method lock() {  b = false ; }

        partial action monitor
        {
               (!b) ; Keyboard.getword(w) --> { b = (pword.equals(w)) ; }
           [+] ( b) ; Keyboard.getword(w) --> { Display.put(w) ; }
        }

    } // item ScreenSaver

    ScreenSaver needs Keyboard, Devices ;
    export ScreenSaver ;

 } // module SS
```

Ex. 9: A large example.
Hamming's problem.

```
module Channel {
    local { class Nat { public unsigned int val ; } }

    box NatFifo { ... }  // for elements of type Nat
    box intFifo { ... }  // for elements of type int

    export NatFifo, intFifo ;
}

module Hamming
{
    import Channel ;
    item printq : intFifo ;
    inner printq + {}
    export printq ;
}

module stream
{
    local { class Nat { public unsigned int val ; } }

    import Channel ;
    import Hamming.printq ;
    export printq ;

    inner printq + { put }
    outer printq - { put }

    produce needs consume, mult0, mult1, mult2, merge ;
    consume needs produce, mult0, mult1, mult2, merge , printq ;

    item mult0, mult1, mult2, merge : NatFifo ;

    item produce
    {   local
        {       Nat h[3] , f ;

                Nat min(Nat a,b,c)
                {
                    Nat t ;
                    t.val = (a.val > b.val) ? a.val : b.val ;
                    t.val = (t.val > c.val) ? t.val : c.val ;
                    return t ;
```

```
                }
            }

            init { f.val = 0 ; h[0].val = 0 ; h[1].val = 0 ;
                   h[2].val = 0 ; merge.put(Nat(1)) ;
                 }

            forall i : 0..2
              partial action read
                {
                    (h[i] == 0) ; mult[i].get(h[i]) --> { ; }
                }

            partial action write
            {
                (h[0].val != 0) && (h[1].val != 0) && (h[2].val != 0) -
->
                      { f = min(h[0],h[1],h[2]) ;
                        merge.put(f) ;
                        if (f.val == h[0].val) { h[0].val = 0 } ;
                        if (f.val == h[1].val) { h[1].val = 0 } ;
                        if (f.val == h[2].val) { h[2].val = 0 } ;
                      }
            }
        } // item produce


        item consume
        {
            partial action
            {
                local { Nat g , h ; }

                    ; merge.get(g)  --> { printq.put(g.val) ;

                                          h.val = 2*g.val ; mult[0].put(h) ;
                                          h.val = 3*g.val ; mult[1].put(h) ;
                                          h.val = 5*g.val ; mult[2].put(h) ;
                                        }
            }
        } // item consume
    } // module stream
```

Now any execution that contains the item produce will also contain the items consume, mult0, mult1, mult2, merge and printq.

References

[GJS 96] Gosling, J., Joy, W., Steele G., The Java Language Specification, Addison-Wesley, 1996

[Krüger 96] Krüger, I. H., An Experiment in Compiler Design for a Concurrent, Object-Based Programming Language, M.S.Thesis, Univ of Texas, 1996

[Misra 96] Misra, J., A Discipline of Multiprogramming, unpublished manuscript, 1996 `ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z`

[Misra 94] Misra, J., Closure Properties, unpublished manuscript, 1994 `ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/closure.ps.Z`