A Machine Readable Syntax for AWN Specifications

Rob van Glabbeek
Data61, CSIRO, Sydney, Australia
School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
rvg@cs.stanford.edu

An AWN specification is a mathematical object defined in [arxiv:1312.7645]. This document proposes a formal definition of a machine readable ASCII AWN specification. It unambiguously determines what does and what does not count as an ASCII AWN specification, and maps each of them to an AWN specification.

1 ASCII AWN Files

An *ASCII AWN file* must contains a finite string of *text characters*. There are 95 text characters, namely those ASCII characters numbered 32 to 126, i.e., all ASCII characters except the 33 control characters.

A *raw* ASCII AWN file may additionally contain tabs and linebreaks. A *multi-line comment* in a raw ASCII AWN file is any substring starting with "{*" and ending with the first occurrence of "*}" following this "{*"; a correct raw ASCII AWN file may not have multi-line comments containing further occurrences of "{*", nor occurrences of "{*" that are not followed by "*}". A *1-line comment* in a raw ASCII AWN file is any substring starting with "`--`" and ending at the first linebreak following the "`--`". *Purging* a raw ASCII AWN file consist of first deleting all multi-line comments, and then deleting all single-line comments, including the opening "`--`", and excluding the linebreak. Each raw ASCII AWN file gives rise to an ASCII AWN file, obtained by first purging the comments, and then replacing each tab and linebreak by a space.

Words and Spaces We define a *word* as one of the following nonempty strings of text characters:

- a *name string*: a string of the 62 alphanumeric characters and the following 14 characters:

  `.   _   "   #   $   %   '   /   ?   @   \   ^   ‘   ~`

- an *infix string*: a string of the following 10 characters: `*   +   -   :   <   =   >   !   &   |`
- a string of square opening and closing brackets `[ ]`
- one of the following 6 single-character strings: `{   (   )   }   ,   ;`

Moreover, in a name string of length > 1 ending on a period, the ending period is seen as a single-character word, not part of the rest of the name string. An ASCII AWN file can be been seen as a sequence of words. Spaces are used only to separate words. They are always optional, except when separating two words of the same kind that are not single-character strings. The meaning of an ASCII AWN file remains invariant under contraction of any nonempty sequence of spaces into a single space.

Names We define a *name* as a name string that does not occur in the following list of *keywords*.

```
forall   exists   lambda   .  include   proc   INCLUDES
TYPES  VARIABLES  CONSTANTS  FUNCTIONS  PREDICATES  PROCESSES  ALIASES
```

The strings `!` and `[]` are also allowed as a names. An *infix expression* is an infix string, but excluding the strings `:=` and `!`.

Type Expressions The *type expressions* are defined by the following context-free grammar in BNF form.

$$TE ::= Name \mid (TE\ \texttt{x}\ TE\ \texttt{x} \cdots \texttt{x}\ TE) \mid (TE\ \texttt{->}\ TE) \mid (TE\ \texttt{+->}\ TE) \mid \texttt{Pow}(TE) \mid [TE]$$

Here *TE* is a type expression, and *Name* a name, as defined above, but excluding the names `x`, `!` and `[]`. Brackets may be deleted iff they are determined by context, taking into account that `x` binds stronger than `->` and `+->`. A type expression is *binary* if it has the form (TE_1 `->` TE_2) or (TE_1 `+->` TE_2), where TE_1 has the form (TE_3 `x` TE_4).

An *interpreted type expression* is a type expression seen as a parse tree. The brackets do not occur in this parse tree, but are used to parse unambiguously. Two type expressions are *equivalent* if they give rise to the same interpreted type expression. In this case they differ only in optional spaces and brackets.

Data Expressions The *data expressions* are defined by the following context-free grammar.

$$DE ::= Name \mid (DE\ Infix\ DE) \mid (DE, DE, \cdots, DE) \mid DE(DE) \mid \{DE\} \mid \{Name \mid DE\} \mid$$
$$\{(Name, DE) \mid DE\} \mid \texttt{lambda}\ Name\ .\ DE \mid \texttt{forall}\ Name\ .\ DE \mid \texttt{exists}\ Name\ .\ DE \mid$$

Here *DE* is a data expression, and *Name* a name. *Infix* is an infix expression. All brackets, except the ones in $\{(Name, DE) \mid DE\}$, may be deleted iff they are determined by context, taking into account that juxtaposition (interpreted as function application) *DE DE* binds strongest of all, followed by *DE Infix DE* and then *DE*, *DE*. The last three constructs bind weakest of all. An *interpreted data expression* is a type expression seen as a parse tree.

Variable Expressions The *variable expressions* are defined by the following context-free grammar.

$$VE ::= Name \mid VE\ [DE]$$

Here *VE* is a data expression, *Name* a name, and *DE* a data expression.

Process Expressions The *sequential process expressions* are defined by the following grammar.

$$SPE ::= Name\ (EL) \mid [DE]\ SPE \mid [[VE := DE]]\ SPE \mid (SPE + SPE) \mid$$
$$\texttt{unicast}(DE, DE)\ .\ SPE > SPE \mid \texttt{unicast}(DE)\ .\ SPE \mid \texttt{broadcast}(DE)\ .\ SPE \mid$$
$$\texttt{groupcast}(DE, DE)\ .\ SPE \mid \texttt{send}(DE)\ .\ SPE \mid \texttt{deliver}(DE)\ .\ SPE \mid \texttt{receive}(VE)\ .\ SPE$$

Here *SPE* is a sequential process expression, *Name* a name, *VE* a variable expression, *DE* a data expression, and *EL* a comma-separated list of data expressions. In case *EL* is the empty list, its surrounding brackets may be omitted. Brackets may be added to facilitate unique parsing, or deleted iff they are determined by context, taking into account that in a *sum* $SPE_1 + SPE_2 + \ldots + SPE_n$ brackets associate to the right. An *interpreted sequential process expression* is a sequential process expression seen as a parse tree.

Blocks A file is a correct ASCII AWN file iff it is a sequence of *ASCII AWN blocks*, defined below. There are seven kinds of blocks: *inclusion blocks*, *type blocks*, *variable blocks*, *constant blocks*, *function blocks*, *process blocks* and *alias blocks*.

An include block consists of the keyword `INCLUDES:` followed by a sequence of URLs or relative path names from the current file or directory to another file or directory. If there is only one URL or relative path name in the block, the keyword may also be `include`.

A type block consists of the keyword `TYPES:` followed by a sequence of type declarations. Each type declaration has the shape *Name* or *Name* = *TE*, where *Name* is a name, but not `x`, and *TE* a type expression.

A variable block consists of the keyword `VARIABLES:` followed by a sequence of variable declarations. Each variable declaration has the shape *Name*: *TE*. Alternatively, a variable block may consist of the keyword `VARIABLES:` followed by a sequence of declarations of the form *TE list*, where *TE* is a type expression and *list* a comma-separated list of names. A single declaration *TE list* is equivalent to the list of declarations *Name*: *TE*, containing one element for each *Name* occurring in *list*.

A constant block consists of the keyword `CONSTANTS:` followed by a sequence of constant declarations. A constant declaration has the shape *Name*: *TE*. Alternatively, a constant block may consist of the keyword `CONSTANTS:` followed by a sequence of declarations of the form *TE list*, where *TE* is a type expression and *list* a comma-separated list of names. The semantics of this alternative form is as for variables.

A function block consists of the keyword `FUNCTIONS:` followed by a sequence of function declarations. A function declaration has the shape *Name*: *TE*, where *Name* is a name and *TE* a type expression. It may also have the form *Infix*: *BTE*, where *Infix* is an infix expression and *BTE* a binary type expression.

A process block consists of the keyword `PROCESSES:` followed by a sequence of process declarations. If there is only one process declaration in the block, the keyword may also be `proc`. Each process declaration has the shape *Name*(*list*) := *SPE*, where *Name* is a name, *list* is a comma-separated list of names, and *SPE* a sequential process expression. In case the list is empty, the brackets may be omitted.

An alias block consists of the keyword `ALIASES:` followed by a sequence of alias declarations. An alias declaration has the shape *Name* := `"`*DE*`"`, with *Name* a name and *DE* a data expression. It may also have the shape *Name* := `"`*list*`"`, with *list* a comma-separated nonempty list of names.

Each AWN file can be read in only one way as a sequence of blocks. Moreover there is only one way to split a block into declarations. For all declarations, *Name* or *Infix* is called the *declared name*.

2 The Interpretation of ASCII AWN Files and Directories

The *interpretation* of a type declaration, called an *interpreted* type declaration, is either a name, or a pair of a name and an interpreted type expression. Likewise, the interpretation of a variable, constant, function or alias declaration is a pair of a name and a interpreted type expression. The meaning of a process declaration is a triple of a name, a list of names, and an interpreted process expression.

An *AWN ASCII* directory is a directory, such that all files in it with the file-extension ".awn" are valid raw AWN-ASCII files as defined above. Such a directory can equivalently be seen as an AWN ASCII file containing only include blocks, namely one for each ".awn" file in it.

Now define a partial order "prior" on files and directories, namely the least one such that file A is prior to file B if B is a (raw) AWN ASCII file with an include block in it pointing to A. A raw AWN-ASCII file A_0 is *well-founded* if there is no infinite sequence A_1, A_2, ... such that A_{i+1} is prior to A_i; moreover all files prior to A_0 should be correct (raw) AWN ASCII files as well.

The interpretation of a well-founded (raw) AWN ASCII file or directory B is defined with induction on the prior relation. It is a 6-tuple (T,V,C,F,S,A) of sets of interpreted type-, variable-, constant-, function-, sequential process and alias declarations. By induction assume that for each include block occurring in B such a 6-tuple has been generated for the AWN ASCII file or directory pointed to by the include block. Now take the componentwise union of these tuples and add all the interpreted declarations contributed in the present file. Note that this union need not be disjoint.

Our use of *interpreted* declarations allows regarding two declarations to be the same, in taking this union, even when as strings they differ slightly in spaces and brackets.

3 ASCII AWN Specifications

A (raw) ASCII AWN file or directory constitutes an ASCII AWN Specification iff it is well-founded and its interpretation has no overloading, or only innocent overloading, and it type checks, as defined in Section 5. *Overloading* occurs if the file's interpretation as a 6-tuple of declarations contains two declarations with the same declared name. A type declaration with and without an associated type expression does not count as overloading, and is equivalent to just the one with the associated type expression. A pair of declarations with the same declared name is considered an *innocent* case of overloading if one is a function declaration with a type TE_1 -> TE_2 or TE_1 +-> TE_2, and the other a variable, constant or function declaration with a type not of the form TE_1 -> TE_3 or TE_1 +-> TE_3.

4 Default Types and Functions

The following types, variable, constants and functions are by default part of any ASCII AWN specification, even if not explicitly declared. It is as if they are always included through an implicit include block. The type `TIME` and variable `now` only occur if the AWN specification is in fact a T-AWN specification.

```
TYPES:
  Bool               -- The Booleans
  DATA               -- Application layer data
  MSG                -- Messages
  IP                 -- IP addresses, or any other node identifiers
  TIME               -- Time values, typically the integerts with infinity

VARIABLES:
  now:               TIME

CONSTANTS:
  true:              Bool
  false:             Bool

FUNCTIONS:
  !   :              Bool -> Bool                -- negation
  &   :              Bool x Bool -> Bool         -- conjunction
  |   :              Bool x Bool -> Bool         -- disjunction
  ->  :              Bool x Bool -> Bool         -- implication
  <-> :              Bool x Bool -> Bool         -- bi-implication
  newpkt:            Data x IP -> MSG            -- message from application layer
```

Additionally, for each type *Type* there are functions

```
   =   :             Type x Type -> Bool         -- equality
   !=  :             Type x Type -> Bool         -- inequality
   isElem:           Type x Pow(Type) -> Bool    -- is an element of
```

The infix operators `->` and `<->` bind weakest of all, then `&` and `|`, then `=` and `!=`, and then all others.

5 Type Checking

We interpret the ASCII AWN file as a 6-tuple (T,V,C,F,S,A), as described in Section 6. T can be seen as a set of declared type names (the *declared types*), together with a partial function of some of these types to type expressions. Likewise V can be seen as a total function from a set of declared variable names (the *variables*) to type expressions. Similarly, C is a total functions from a set of declared constant names (the *constants*) to type expressions, A is a total function from a set of declared alias names (the *aliases*) to the union of the set of data expressions and the set of lists of names, and S is a total function from a set of declared process names (the *processes*) to pairs of lists of names and process expressions. Let a *data alias* be an alias that maps to a data expression, and a *list alias* one that maps to a list of names.

The requirement regarding overloading implies that the sets of declared types, variables, constants, processes and aliases are disjoint. The set F can be seen as a total function from the function declarations (the *functions*) to the type expressions. There can be multiple functions with the same declared name, and this name can also be used as a constant or a variable.

An ASCII AWN file *type checks* iff the following conditions are met.

1. Type expressions are associated to some declared types, and to all variables, constants and functions. We require that all names that occur in these type expressions are themselves declared types. In this case, the type expression constitutes the *type* of the variable, constant, function or type expression.
2. Call a type declaration type_1 to be *prior* to a type declaration type_2, if type_1 occurs in the type of type_2. We rule out circular type declarations of type_1 being prior to type_2 being prior to $\ldots \mathit{type}_n$ being prior to type_1.

   A declared type name with an associated type can be seen as an abbreviation. It can always be interpreted by substituting for it its associated type.
3. Data expressions occur in process expressions and are associated to aliases. All names occurring in a data expression must be variables, constants, function names or data aliases. Moreover, functions of a type type_2 `->` type_3 or type_2 `+->` type_3 may only occur in the role of DE_1 in a subexpression $\mathit{DE}_1(\mathit{DE}_2)$. All infix expressions occurring in a data expression must be functions names.
4. Call a data alias alias_1 to be *prior* to a data alias alias_2, if alias_1 occurs in the data expression associated to alias_2. We rule out circular type declarations of alias_1 being prior to alias_2 being prior to $\ldots \mathit{alias}_n$ being prior to alias_1.
5. List aliases must map to repetition-free lists of variables.
6. The list of names associated to a process expression must be a list of variables and list aliases. After substituting the associated list of variables for each data alias, the resulting list of variables must be repetition-free.
7. The following algorithm tells when type checking fails for a data expression or data alias, and associates a type to every data expression and data alias that type checks correctly. It operates by induction on the prior relation, and by structural induction on data expressions. If type checking fails on a subexpression of a data expression, then it also fails on the whole data expression.
   - The type of a variable or constant is given by the components V and C of the interpreted AWN ASCII file.
   - A function name occurrence that is not DE_1 in a subexpression $\mathit{DE}_1(\mathit{DE}_2)$, denotes, by the rules on overloading and Clause 3 above, a unique function. Its type is given by the component F of the interpreted AWN ASCII file.
   - The type of a data alias is the type of the associated data expression, provided the latter type checks correctly. Otherwise type checking fails for this data alias.

- An expression (DE_1 *Infix* DE_2) where $type_1$ is the type of DE_1 and $type_2$ is the type of DE_2, type checks correctly iff the function *Infix* has a type ($type_1$ x $type_2$) `->` $type_3$ or ($type_1$ x $type_2$) `+->` $type_3$. In that case the type of the expression is $type_3$.
- An expression (DE_1, DE_2, $\cdots$, DE_n) where $type_i$ is the type of DE_i for $i = 1,\ldots,n$, has type ($type_1$ x $type_2$ x $\cdots$ x $type_n$).
- An expression $DE_1(DE_2)$ where $type_1$ is the type of DE_1 and $type_2$ is the type of DE_2, type checks correctly iff $type_1$ has the shape $type_2$ `->` $type_3$ or $type_2$ `+->` $type_3$. In that case the type of the expression is $type_3$. In the special case that DE_1 is a function name, there is at most one function with that name that has a type for the form $type_2$ `->` $type_3$ or $type_2$ `+->` $type_3$; this is the function denoted by that name.
- An expression $\{DE\}$ where *type* is the type of *DE*, has type `Pow`(*type*). In the special case that *DE* has the shape *Name* | *DE* it is always seen as an expression of the form below.
- An expression $\{Name \mid DE\}$ type checks if *Name* is a variable name, say of type *type*, and *DE* has type `Bool`. In that case the type of the expression is `Pow`(*type*).
- An expression $\{(Name, DE_1) \mid DE_2\}$, where $type_1$ is the type of DE_1, type checks if *Name* is a variable name, say of type $type_0$, and DE_2 has type `Bool`. In that case the type of the expression is $type_0$ `+->` $type_1$.
- An expression `lambda` *Name* . *DE*, where $type_1$ is the type of *DE*, type checks if *Name* is a variable name, say of type $type_0$. In that case the type of the expression is $type_0$ `->` $type_1$.
- An expression `forall` *Name* . *DE* or `exists` *Name* . *DE* type checks if *Name* is a variable name and *DE* has type `Bool`. In that case the type of the expression is `Bool`.

8. A variable expression type checks correctly iff all data expression occurring in it type check correctly, and the following requirements are met.
   - In an expression *Name*, *Name* is a variable. The type of the expression is the type of *Name*.
   - In a subexpression *VE*[*DE*], *DE* a data expression, say of type $type_1$. The expression type checks iff *VE* is a variable expression of type (DE_1 `->` DE_2) or (DE_1 `+->` DE_2). The type of the expression is $type_2$.
9. A process expression type checks correctly iff all data expression occurring in it type check correctly, and the following requirements are met.
   - In each subexpression [*DE*] *SPE*, the data expression *DE* is of type `Bool`.
   - In each subexpression [[*VE* := *DE*]] *SPE*, *VE* is a variable expression, of the same type as *DE*.
   - In each subexpression `unicast`(DE_1, DE_2) . SPE_1 > SPE_2 or `unicast`(DE_1, DE_2) . *SPE*, the data expression DE_1 is of type `IP`, and DE_2 is of type `MSG`.
   - In each subexpression `groupcast`(DE_1, DE_2) . *SPE*, the data expression DE_1 is of type `Pow(IP)`, and DE_2 is of type `MSG`.
   - In each subexpression `broadcast`(*DE*). *SPE* or `send`(*DE*) . *SPE*, the data expression *DE* is of type `MSG`.
   - In each subexpression `deliver`(*DE*) . *SPE*, the data expression *DE* is of type `DATA`.
   - In each subexpression `receive`(*VE*) . *SPE*, *VE* is a variable expression of type `MSG`.
10. When a process name maps to a list *LE* of variables and a process expression *SPE*, all variable occurrences in *SPE* must be bound. An occurrence of a data variable in *SPE* is *bound* if it occurs in *LE*, or is a variable *Name* occurring in a subexpression `receive`(*Name*).SPE', a variable *Name* occurring in a subexpression [[*Name* := *DE*]] SPE', or an occurrence in a subexpression [*DE*] SPE' of a variable occurring free in *DE*. Here an occurrence of a data variable *Name* in a data expression

is *free* if it lays not in subexpression of one of the five forms $\{Name \mid DE\}$, $\{(Name, DE) \mid DE\}$, `lambda` *Name* . *DE*, `forall` *Name* . *DE* or `exists` *Name* . *DE*.

6 The Data Structures of AWN

The process algebra AWN was introduced in [**?**, **?**], which states:

> The internal state of a process is determined, in part, by the values of certain data variables that are maintained by that process. To this end, we assume a data structure with several types, variables ranging over these types, operators and predicates. First order predicate logic yields terms (or *data expressions*) and formulas to denote data values and statements about them.[1] Our data structure always contains the types `DATA`, `MSG`, `IP` and $\mathscr{P}(\texttt{IP})$ of *application layer data*, *messages*, *IP addresses*—or any other node identifiers—and *sets of IP addresses*.

Here a “data structure” refers to a pair $(\Sigma, \mathscr{A})$ of a signature Σ and a Σ-algebra $\mathscr{A}$. Such a data structure figures as a parameter in the definition of AWN. Here a *signature* is a list of types and a list of variable declarations, constant declarations, function declarations and predicate declarations, each consisting of a name and a type. The Σ-algebra $\mathscr{A}$ describes the *semantics* of the declared constants, functions and predicates. This semantics is not addressed in the present paper.

In the present paper we use a common encoding of predicates as functions in the Booleans. This makes it unnecessary to treat predicates separately. Consequently, a formula becomes a data expression of type `Bool`.

Originally, Σ and $\mathscr{A}$ were meant to be a first-order signatures and algebras; however, all of [**?**, **?**] is consistent with taking Σ and $\mathscr{A}$ to be higher-order signatures and algebras—in which case “First order predicate logic” should be read as “Higher order predicate logic”. In first-order logic, variables and constants must have a basic type, rather than a complex type build using the type constructors `x`, `->`, `+->`, `Pow` and `[ ]`. Moreover functions must have a type $type_1$ `x` $type_2$ `x` $\dots$ `x` $type_n$ `->` $type_0$ where the $type_i$ are basic types. Finally, first-order logic lacks the constructs $\{Name \mid DE\}$, $\{(Name, DE) \mid DE\}$ and `lambda` *Name* . *DE* for building data expressions.

The current paper formalises higher-order signatures. These are the components T, V, C and F of an ASCII AWN specification (T,V,C,F,S,A). The aliases A are merely a simply abbreviation mechanism.

The data structures that are allowed as parameter of AWN are required to contain the types `DATA`, `MSG`, `IP` and $\mathscr{P}(\texttt{IP})$. In case a higher-order data structure is meant, the requirement on $\mathscr{P}(\texttt{IP})$ is redundant, as such a type comes for free with the type `IP`.

In [**?**, **?**] AWN is applied to model the Ad hoc On-Demand Distance Vector (AODV) protocol [**?**]. For this purpose, a specific data structure $(\Sigma, \mathscr{A})$ is chosen as parameter. It features types `IP`, `SQN`, `K`, `F`, $\mathbb{N}$, `R`, `RT`, `RREQID`, `P`, `DATA`, `STORE`, `MSG`, `[MSG]`, $\mathscr{P}(\texttt{IP})$, $\texttt{IP} \rightharpoonup \texttt{SQN}$ and $\mathscr{P}(\texttt{IP} \times \texttt{RREQID})$. In [**?**] this list of types is partitioned into twelve basic types and four higher-order types, obtained by the constructions [`TYPE`], $\mathscr{P}(\texttt{TYPE})$, $\texttt{TYPE}_1 \rightharpoonup \texttt{TYPE}_2$ and product $\times$. This matches the type constructors of Section 1. Variables are declared for the above-mentioned types only; additionally there are (partial) functions $f : T_1 \times \cdots \times T_n \to T$ and $f : T_1 \times \cdots \times T_n \rightharpoonup T$, with $T_1, \dots, T_n, T$ chosen from the types above. The type `R`, although introduced as a basic type, could equally well have been cast as $\texttt{IP} \times \texttt{SQN} \times \texttt{K} \times \texttt{F} \times \mathbb{N} \times \texttt{IP} \times \mathscr{P}(\texttt{IP})$; this would eliminate the need to declare the function $(\_,\_,\_,\_,\_,\_,\_) : \texttt{IP} \times \texttt{SQN} \times \texttt{K} \times \texttt{F} \times \mathbb{N} \times \texttt{IP} \times \mathscr{P}(\texttt{IP}) \to \texttt{R}$.

[1] As operators we also allow *partial* functions with the convention that any atomic formula containing an undefined subterm evaluates to `false`—footnote added in [**?**].

Higher-order terms and formulae do occur in [**?**, **?**]; in case first-order predicate logic is meant, these terms require an encoding into first-order form, which is not explicitly given in [**?**, **?**]. Higher-order terms occur for instance in [**?**, Process 1, Line 18], which is [**?**, Process 5, Line 16]:

$$\mathtt{dests} := \{(\mathtt{rip},\mathtt{inc}(\mathtt{sqn}(\mathtt{rt},\mathtt{rip}))) \mid \mathtt{rip} \in \mathtt{vD}(\mathtt{rt}) \wedge \mathtt{nhop}(\mathtt{rt},\mathtt{rip}) = \mathtt{nhop}(\mathtt{rt},\mathtt{oip})\}.$$

Here `dests` is a variable of type $\mathtt{IP} \rightharpoonup \mathtt{SQN}$, so the right-hand side must be a term of this type. In fact, it is constructed by the seventh clause for data expressions in Section 1. There appears to be no direct way to translate this higher-order term into a first-order term.

The eighth clause for data expressions in Section 1—λ-abstraction—is not used in [**?**, **?**], but set comprehension, partial function application and quantification occur in [**?**, Process 1, Line 20], which is [**?**, Process 5, Line 19]:

$$\mathtt{pre} := \bigcup\{\mathtt{precs}(\mathtt{rt},\mathtt{rip}) \mid (\mathtt{rip},*) \in \mathtt{dests}\}$$

which is a shorthand for

$$\mathtt{pre} := \bigcup\{\mathtt{y} \mid \exists\mathtt{rip}\,\big(y = \mathtt{precs}(\mathtt{rt},\mathtt{rip}) \wedge \exists\mathtt{rsn}\big((\mathtt{rip},\mathtt{rsn}) \in \mathtt{dests}\big)\big)\}$$

and, by the syntax of Section 1, formally written as

$$\mathtt{pre} := \mathtt{UNION}\,\big(\{\mathtt{y} \mid \mathtt{exists}\;\mathtt{rip}.\,\big(\mathtt{y} = \mathtt{precs}(\mathtt{rt},\mathtt{rip}) \wedge \mathtt{exists}\;\mathtt{rsn}.\big(\mathtt{dests}(\mathtt{rip}) = \mathtt{rsn}\big)\big)\}\big)$$

using that `dests` is of type partial function, rather than set. Here `rt` is a variable of type `RT`, `rip` of type `IP` and `rsn` of type `SQN`; `precs` is a function symbol of type $\mathtt{RT} \times \mathtt{IP} \to \mathscr{P}(\mathtt{IP})$ and `pre` and `y` variables of type $\mathscr{P}(\mathtt{IP})$. The function `UNION` used here is of type $\mathscr{P}(\mathscr{P}(\mathtt{IP})) \to \mathscr{P}(\mathtt{IP})$.

7 AWN Specifications

Besides the choice of a data structure, described above, an application of AWN, as described in [**?**, **?**] is parametrised by the choice of a collection of process names with defining equations. This is formalised exactly by component S of an ASCII AWN specification (T,V,C,F,S,A).