IMPACT Agent definition syntax.

3.0 Introduction: An IMPACT agent definition is a collection of specifications (identification, connection, type, function, action, program, action-constraint, and integrity-constraint) necessary to describe the target agents behavior. Agent identification and connection criteria are typically specified using the AgentDE graphical user interface (GUI) controls to select and specify the necessary input parameters. As for the type, function, action, program, action-constraint, and integrity-constraint definitions (hereafter called parseable components), the AgentDE provides a series of parsers which render instantiated objects from the definition text input. As mentioned previously, the AgentDE lower tab-control panels provide the agent specification input controls; Here you should note that for every parseable component, there is a corresponding definition entry page which includes a text entry area and two buttons (parse and clear). Simply select the desired input page (bringing the corresponding control page forward), enter the text into the edit window, and then invoke the parser by selecting the "Parse" button.

If the text parses correctly, the system will generate the object and insert it into the agent instantiation (the IAD_MetaData class). If, however, the system detects an error, an dialog box will appear with the corresponding error text.

NOTE: **The connection interface (the *Java* `IAD_Connect` class) provides the ability to import pre-defined connection type and function code-call definitions**. That is to say that if the connection developer defined types and/or code-call functions for the connection interface, then your agent instantiation automatically inherits such type and function definitions when you select that connection for use in your agent instantiation – You need not define such connection specific types or functions manually.

3.1 General parser syntax: This document uses a Bachaus-Naur like format to describe syntax used in the various agent definition components. Some tokens, such as "ID", "INT", and "STRING", apply universally and thus are presented here before anything else.

```
ID ::= (‘a’..‘z’ | ‘A’..‘Z’)
       (‘a’..‘z’ | ‘A’..‘Z’ | ‘0’..‘9’ |
‘_’)*

RECID ::=  ID (  ‘.’  ID  )+

INT_VALUE ::=  (  ‘0’..‘9’ )+

INT ::=  ( ‘+’  | ‘-’  )?   INT_VALUE

REAL ::=  INT  ‘.’  INT_VALUE

STRING ::=  ‘"’  (  ~‘"’  )*  ‘"’

FILE ::=  ‘\’’  (  ~‘\’’  )*  ‘\’’

AND ::= ‘&’

OR ::=  ‘|’

NOT ::=  ( ‘!’  | ‘~’)

BOOL ::=  ‘$’ ( “true” | “false”) ‘$’
```

Fig 3.1: Base-level tokens.

From the base-level token syntax (see Fig 3.1) we see that an ID token must start with an alpha character and can be followed by zero or more alpha-numeric or underscore characters.

‘a’..’z’	denotes the span of characters from ‘a’ to ‘z’
(x | y)	denotes that item x or y applies
()*	denotes that the enclosed items occur zero or more times.
()+	denotes that the enclosed items occur one or more times.
()?	denotes that the enclosed item is optional.
STRING	denotes text inclosed in double quotes.
FILE	denotes text inclosed in single quotes.

3.2 IMPACT Agent type definitions: Agent type definitions describe storage and argument data structure schemas used in all the higher-level agent definitions (such as connection code-call functions, actions, and action procedures). Current agent type definitions occur in two flavors, data type sets and records; Simply speaking, data sets are bound by curly brackets `“{}”`, data records are bound by angle brackets `“<>”`, and record field names are separated from schema names by the forward slash `‘//’` character (See Syntax3.2 for the **current** type grammar).

The type syntax allows us to define simple as well as nested hierarchy type definitions. Of note, the “base_type” tokens can reference either standard simple types (i.e. integer, real, char, boolean, file, and string), special simple types (i.e. any, object – to be discussed later), or they can reference any previously defined type definition by name (the system attempts to locate and set the necessary sub-type references during the parsing phase).

```
IMPACT agent type definition ::=
        type_node ‘;’ EOF

type_node ::= ( record_node | set_node | base_type
)

set_node ::=
    SchemaTok:ID ‘{‘ ( record_node | base_type )
‘}’

record_node ::= SchemaTok:ID
    ‘<’
    NameTok:ID ‘/’ type_node
    ( ‘,’ NameTok_N:ID ‘/’ type_node )*
    ‘>’

base_type ::=
    “integer” | “real” | “char” |
    “boolean” | “file” | “any” |
    “object” | “string” | SchemaTok:ID
```

Fig 3.2: Agent type grammar.

Suppose for example we're defining an agent which deals with route planning. The first thing we might want to define is a type for the set of points which describing a travel path (see Fig 3.3). Select the type edit control page, enter the definition text into the edit window and select (click on) the "Parse" button. After parsing the type definition text you should note that the text entry area has cleared. This, and the status bar entry '*Success – "PointSet" type added to table*', indicates that the defininition parsed successfully. To view the newly rendered type definitions, select the upper AgentDE tab-control panel marked "UserDef T&F Lib" (user defined type and function definition library). Here you should now see two new type entries in the left-hand list box (`PointSet` and `PointRecord`); The example text actually defined two types, a set type and a record type, in one nested definition.

```
PointSet
{
   PointRecord
   <
      X/integer,
      Y/integer
   >
};
```

Fig 3.3: PointSet type.

Continuing with this example, a typical route planning algorithm might render a set of one or more routes (for client selection). The Fig 3.4 lists the `RouteRecord` type, one field of which references the previously defined `PointSet` type, and the `RouteSet` type as a set of `RouteRecords`:

```
RouteSet
{
   RouteRecord
   <
      RouteCost/double,
      RouteLength/integer,
      Route/PointSet,
   >
};
```

Fig 3.4: RouteSet type..

The type definitions from figures 3.3 and 3.4 ultimately rendered four type objects from two input text definitions. Using a type definition variation, you could have just as easily have defined all four from one text definition as depicted in Fig 3.4. Please note that here the `Route/PointSet{}` entry defines not only the field and schema name (separated by a forward slash), but the underlying schema as well.

```
RouteSet
{
   RouteRecord
   <
      RouteCost/double,
      RouteLength/integer,
      Route/PointSet
      {
         PointRecord
         <
            X/integer,
            Y/integer
         >
      },
      RouteScript/file
   >
};
```

Fig 3.5: Four types in one definition.

3.3 IMPACT Agent function definitions provide code-call prototypes (see Fig 3.7 for precise grammar) for methods invoked through a specified connection interface (IMPACT invokes code-calls through connection interfaces). The AgentDE uses these prototypes later in the compilation and run-time process to validate parameter and return-type compatibility. Simply speaking, an IMPACT agent function definition lists the connection alias, domain and function names, the parameter list (may be void – similar to the *C* language, void functions have no parameters), and the return type as follows:

```
AliasName –-> DomainName:FunctionName
(Param1/Type1, Param2/Type2, ···,ParamN/TypeN):ReturnName/ReturnType;
```

The following example shows a function definition used to invoke the Hermes oracle:project method via an RHermes connection (aliased as "Jilad"):

```
Jilad-->oracle:project( SrcTable/file, ConnectInfo/string,
ProjectFlds/string): ret/APS_LOC_2D;
```

This function takes three input parameters (a file and two strings) and returns an argument of type "APS_LOC_2D" (a set of string records – see Fig 3.6).

```
APS_LOC_2D
{
   LOC_Recd1
   <
      LOC/string
   >
};
```

Fig 3.6: APS_LOC_2D type.

Note: As mentioned previously, if the agent developer uses a connection interface which does not provide a method listing, then they must define the types and/or method prototypes for all the types or methods they plan to invoke through the connection within the agent instantiation. If, however, the connection does provide a method listing (the preferable design), then the agent instantiation inherits those type and function definitions from the connection reference; The agent developer then simply references the desired code-call methods within the program.

```
IMPACT Agent function definition ::=
   ConnAliasTok:ID "-->"
   DomainTok:ID ':' FunctionTok:ID
'(' paramList "):" returnType ';' EOF

paramList ::=
   paramType ( ',' paramType )*  |
"void"

paramType ::=
   FnameTok:ID '/' typeName

returnType ::=
   returnParamType | "void"

returnParamType ::=
   ( "polymorphic" | FNameTok:ID '/' typeName
)

typeName ::=
   ( NameTok:ID | "integer" | "real" |
      "char" | "boolean" | "file" |
      "any" | "object" | "string" )
```

Fig 3.7: Agent connection code-call function grammar.

3.4 IMPACT Agent action definitions (see Fig 3.10 for precise grammar) map an agents inner state to its behavior with the outside world (i.e. users, hardware, and other agents). As depicted in section 1, figure 1, action procedures (see `IAD_ActionExe` methods), represent the "data-out" side of an agent. Similar to *C* language function prototypes, an action definition specifies the action name and result data parameters such as `renderMessage( SzMessage/string)`, but not its underlying implementation. Moreover, action definitions can include **Pre**Condition, **Add**, and **Del**ete agent state conditions, and commonly detail action procedure mappings, as necessary, over the prescribed result data set to render the appropriate output. The action procdure library currently includes methods such as createFile, appendFile, mailFile, and sendMessage. Alternately, actions with out an executable component, so called "No Operation" (NoOp) actions, are useful as a shorthand for wrapping code-call subroutines and inserting data facts. During the compilation / testing phase, constants and/or program entry rule bodies for such "NoOp" actions are actually in-line substituted into the unfolded agent program and then removed from the action list.

Note: **The current IMPACT implementation ignores action Precondition, Add, and Delete state condition specifications**; These control measures will be supported in a (near) future version.

When designing an agent, perhaps the first step should be to decompose the problem into appropriate sub-tasks; To send an email message, for example, you must first create a data file, then append data to it, and finally send the file to the mail server. In this example, a list of defined actions (with action procedure mappings) might include CreateMailFile, AppendMailFile, and SendMailFile. With the base set of actions defined, you might then consider sets of facts and subroutines useful in the agent design.

```
MailLocTotalsFile( FnSource/file, SzTo/string);;;;
--> executes
{
   mailFile
   (
      "", FnSource,
      SzTo, "demo@jilad.cs.umd.edu",
      "LocTotals agent generated Army War Reserves
data.",
      "londo.cs.umd.edu"
   )
};
```

Fig 3.8: MailLocTotalsFile action definition.

3.4.1: Action definition example 1 (Fig 3.8) lists the `MailLocTotalsFile` action whose status set will contain a set of data parameters (`FnSource` source file, and `SzTo` address string elements). It further specifies the mapping of those parameters into the `mailFile` action procedure which will fire after IMPACT generates action status sets for all the applicable actions.

Note: Selecting the AgentDE `ActionExe Lib` control tab (in top panel section) prompts the system to display the control page which lists the currently available action procedures. Selecting the listed procedure entries (click on a listed text entry), in-turn, prompts the controls to display information about that action procedure.

The mailFile action execution method (see ActionExe Lib tab entry) requires six input parameters:

FSzSrcPath/file_or_string (it accepts either a file or string type),
FSzSrcFile/file_or_string,
SzTo_Address/string,
SzFromAddress/string,
SzSubject/string,
SzSMTP_Host/string

In this action definition, we can see that the FnSource action status set data parameter was mapped to the mailFile FSzSrcFile input parameter, and the SzTo action status set data parameter was mapped to the mailFile SzTo_Address input parameter. All other mailFile input parameters were defined with constants.

```
LocTotals (SzLOC/string,  D_AuthQty/real,
D_OnHand/real);;;;
--> NoOp;
```

Fig 3.9: LocTotals action definition.

3.4.3 Action definition example 2 (Fig 3.9) defines the LocTotals action whose status set will contain a set of data parameters (`SzLOC` string, `D_AuthQty` real, and `D_OnHand` real elements). This action, however, cites **`NoOp`** as it's used within the program to define a subroutine code block which ultimately gets in-line substituted into the unfolded program.

```
IMPACT Agent action definition ::=
    actionName ';' precondition ';' add ';' delete ';'
    "-->" exeModuleActionCall ';' EOF

actionName ::= NameTok:ID '(' paramList ')'

exeModuleActionCall ::= exeActionCall | "NoOp"

exeActionCall ::= "executes" '{' NameTok:ID '(' actionArgList ')' '}'

actionArgList ::= actionArg ( ',' actionArg )* |

actionArg ::= VarTok:ID | RvarTok:RECID | StringTok:STRING | IntegerTok:INT |
    RealTok:REAL | CharTok:CHAR | FileTok:FILE | BooleanTok:BOOL

paramList ::= formalParam ( ',' formalParam )* | "void"

formalParam ::= FnameTok:ID '/' simpleType

simpleType ::= NameTok:ID | "integer" | "real" | "char" | "boolean" |
    "file" | "any" | "object" | "string"

precondition ::= codeCallCondition

add ::= codeCallCondition

delete ::= codeCallCondition

codeCallCondition ::=
    ( NOT )? simpleCodeCallCondition
    (
        AND ( NOT )? simpleCodeCallCondition
    )*

simpleCodeCallCondition ::= codeCallAtom |
    ( "<" | "<=" | ">" | ">=" | "=" | "<>" )
    '(' argument ',' argument ')'

codeCallAtom ::= "in" '(' argument ',' codeCall ')'

codeCall ::= DomainTok:ID ':' FunctionTok:ID '(' argList ')'

argList ::= argument ( ',' argument )*

argument ::= VarTok:ID | RecVarTok:RECID | StringTok:STRING | IntegerTok:INT |
    RealTok:REAL | CharTok:CHAR | FileTok:FILE | BooleanTok:BOOL
```

Fig 3.10: IMPACT Action grammar.

3.5 IMPACT Agent program definitions (one per agent) map one or more facts or rule body definitions to action status atom instantiations (See Fig 3.15 for the precise grammar). Here the agent developer specifies the constants and /or data evaluations necessary to instantiate the action status atom arguments (the output data). A typical program entry specifies the action status atom, a deontic modality (**Do**, **O**bliged, **F**orbidden, **W**aived, **P**ermitted) wrapper containing an action reference, followed by the optional rule body (a collection of connection code-calls and / or comparisons linked by the ampersand). Program entry action status atom references are separated from the (optional) rule bodies by the "**:-**" operator and must terminate with a period ('**.**'). Finally, the agent program definition must terminate with a semicolon ('**;**') character following the last entry definition. Of note, "facts" are special cases as they typically map constants directly to an action status atom; moreover, facts need not include a rule body (see 3.11, first three program entries)..

3.5.1 Run time evaluation: The current rule-body execution engine follows a stack model. That is to say that at evaluation time, the system processes every action/rule body definition for which the pre-condition state is true (omission of the precondition, currently the default mode, means that the action/rule bodies get evaluated every time). Rule body execution starts with the top rule and continues sequentially through the listed rules (on success the system pushes the rule body element result on the data stack) until either the final rule succeeds, and the action status atom is instantiated from the current stack data, or one of the rules fails (the action status atom is not instantiated for those states).

```
// Simple "ActionName" facts:
//
Mode( ActionName( "Alpha", 5, •••, "Zulu")).
Mode( ActionName( "Bravo", 6, •••, "Zulu")).

// A conditional  "ActionName" fact:
//
Mode( ActionName( "Charlie", 7, •••, "Zulu")) :-
   in( Q, Alias1->Domain_A:Function_1( ))&
   >( Q, 10) &
   <( Q, 20).

// An "ActionName"rule body; The output parameters are
// rendered from data code-calls and constraints
//
Mode( ActionName( OutParam1, OutParam2, •••,
OutParamN)) :-
   in( X, Alias2->Domain_B:Function_1( "Alpha", 5))&
   in( Y, Alias3->Domain_C:Function_1( "Zulu", 2.3,
"Fox"))&
   =( X.SubField1, Y.SubField3)&
        •••
   in( Z, Alias3->Domain_C:Function_2( Y.SubFld1))&
   =( OutParam1, Z.SubField1)&
   <( 5, Z.SubField2) &
   =( OutParam2, Z.SubField3)&
        •••
   =( OutParamN, Z.SubFieldN).
;
```

Fig 3.11: Generic program example.

3.5.2 Comparison operators (listed in Fig 3.12) are common elements of program entry rule bodies. The comparison syntax is a bit backwards (i.e. Reversed Polish Notation)

Operator `'('` Argument1 `','` Argument2 `')'`

Less-Than		<
Less-Than or Equals		<=
Equals	=	
Greater-Than or Equals	>=	
Not Equal		<> or !=

Fig 3.12: Comparison operators.

...but the result is as expected; That is to say that `<(` Value1, Value2`)` evaluates true if Value1 is less than Value2.

3.5.3 The `"in"` operator, which wraps code-call specifications, can functionally be thought of as both as a membership function and a result set assignment reference. That is to say that the evaluation of `"in( X,` AliasName`->`DomainName`:`FunctionName`())"` would be true if the domain function succeeds and either:

1) **X** is a variable -- It then gets assigned the domain function result (set or value).

2) **X** is instantiated (as a value or previous result set reference), and it's a member of the current domain function code-call result set. **Of note, the current implementation does not handle this second case; This, however, is easily remedied by including some additional comparison elements in the rule body.**

3.5.4 Basic Math functions: The current IMPACT implementation provides a "Local" connection module which implements a library of commonly used domain functions (math domain is one of these).

3.5.5 Special operators:

3.5.5.1 Connection code-call caching: Agent programs often include a few rule-body elements which are both costly and execute frequently. IMPACT currently supports caching on specific in operator rule-body element via the cache modifier. The following program entry rule-body element causes the system to consult the internal result cache, prior to execution, for the specified connection code-call:

`cache` `in( X,` ConnAlias`->`Domain`:`Function`(` Arg1`,` Arg2`, ...,` ArgN`))`

If the cache contains a result/reference for such a code-call, then the result is pushed directly on the stack (execution continues with the next rule body element). If the cache does not contain a reference for the code call, then the code-call get's evaluated, and the result is both pushed onto the stack and referenced into the cache.

3.5.5.2 Argument type-casting: One of the special IMPACT types is called "any" (parameters of type `"any"`, accept assignment of any value). This is used, for example in the `sendMessage` action procedure `A_Data` parameter (Fig 3.13); Note, when executed, the sendMessage action procedure renders a `MessageRecd` (Fig 3.14) object containing the necessary message data which, in-turn, gets inserted into the message queue.

```
sendMessage
(
    SzTgtAgent/string,
    SzCommand/string,
    L_Flags/integer,
    A_Data/any
)
```

Fig 3.13: sendMessage action procedure.

```
MessageRecd
<
    SzTo / string,
    SzFrom / string,
    SzCommand / string,
    SzTimeHack / string,
    L_Flag / integer,
    A_Data / any,
    SzKey / string
>;
```

Fig 3.14: Local.MessageRecd type.

Agents can pass anything through the `A_Data` parameter as the field value. On the receiving end, however, an agent can not simply assign something of type `"any"` to, say, an `integer` variable. To get around this, IMPACT supports some rudimentary type-casting capabilities.

Suppose agent A passed a string, for example, as the data input for a message addressed to agent B. Agent B, in-turn, would then retrieve its messages, and access/assign the data field as a string, via program entry rule-body elements something as follows:

```
     ...
in( MsgRecd, Local–>msgBox:getMessages())&
=( SzData, (string) MsgRecd.A_Data)&
     ...
```

3.5.6 Program unfolding: During the agent testing phase, AgentDE attempts to "Unfold" the program in an attempt to generate a simpler program for run-time evaluation. Not unlike subroutine usage (except that the parameters are explicitly data out), rule bodies can include nested references to other action status atoms. The unfolding process, basically, attempts to remove all such nested action status atom references by replacing such references with their (specially modified) factual parameters and/or program entry rule-body definition as applicable (this process is slightly *similar* to *C* language in-line subroutine substitution).

Of note, agent programs without such action status atom reference nesting do not require unfolding. In these cases, the agent evaluates the action status atom set using the original program.

3.5.6.1 "NoOp" action status atom program references renders a developer's short-cut. As cited previously, the IMPACT action definitions (see X) include either a mapping to an action procedure, or a "NoOp" reference (nothing executes over the action status atom data). This is commonly used to tag an "action", and subsequently it's rule body, as an action which does not do anything to change the agent's state (it does not execute any action procedures which would alter the state data). While this may seem pointless, it does, in-fact, render a short-cut syntax for writing a program rule-body "subroutine" (the developer writes it once and then cites it in many places). At unfolding time, the AgentDE inserts the appropriate factual constants and/or rule-body blocks into the new program, and then, finally, removes the "NoOp" action status atom program entries from the rewritten program.

```
IMPACT Agent program definition ::=
    (  ruleHead  (  ":-"  ruleBody  )?  '.'  )+  ';'  EOF

ruleHead ::=  actionStatusAtom

ruleBody ::=
    ruleEntry  (  ( ',' | '&' )  ruleEntry  )*

ruleEntry ::=
    actionStatusAtom  |  simpleCodeCallCondition  |  cacheBlockReference  |
    NOT  (  actionStatusAtom  |  simpleCodeCallCondition  )

cacheBlockReference ::=
    (
        "cacheBlockAs"  '('
        SzNameTok:STRING  ','  '['  argument  (  ','  argument  )*  ']'
','
        (  NOT  )?  simpleCodeCallCondition
        (  ( ',' | '&' )  (  NOT  )?  simpleCodeCallCondition  )*  ')'
    ) |
    (
        "importCacheBlock" '('  SzNameTok2:STRING ')'
    )

simpleCodeCallCondition ::=
    codeCallAtom  |
    (  "<"  | "<="  | ">"  | ">="  | "="  | "<>" )
    '('  argument ',' argument ')'

codeCallAtom ::=
    (  "cache"  )?  "in"  '('  argument  ','  codeCall  ')'  |
    ( "is"  | "store"  )  '('  fileArgument  ','  codeCall  ')'

codeCall ::=
    ConnAliasTok:ID  "-->"  DomainTok:ID  ':'  FunctionTok:ID  '('  argList
')'

argList ::=  argument  (  ','  argument  )*

argument ::=
    (  '('  (  "integer"  | "real"  | "double"  | "char"  |
"boolean"  |
        "file"  | "any"  | "object"  | "string"  )  ')'  )?
    (
        VarTok:ID  |  RvarTok:RECID  |  StringTok:STRING  |  IntTok:INT  |
        RealTok:REAL  |  CharTok:CHAR  |  FileTok:FILE  |  BooleanTok:BOOL
    )

fileArgument ::= FileTok:FILE

actionStatusAtom ::=
    (  "Do"  |  "P"  |  "F"  |  "W"  |  "O"  )  '('  actionCall  ')'

actionCall ::=  NameTok:ID  '('  actionArgList  ')'

actionArgList ::=  actionArg  (  ','  actionArg  )*
```

3.15: Agent program grammar.

3.6 IMPACT Action constraints are not currently supported; This control feature will be available in a (near) future version.

3.7 IMPACT Integrity constraints are not currently supported; This control feature will be available in a (near) future version.