1.10 CONCLUSION. A BRIEF HISTORY

The history of the current version of USSA goes back to the original version of Pyramid's Reliant system (1989-1991). In that system, hardware and software resources had a predefined number of states (like on-line, off-line, faulted, e.t.c.). Resources moved from state to state according to the hardcoded algorithms. Different resources had different but sometimes similar algorithms. It is not known to the author, who has to get credit for the Reliant design.

At that time, Lily Hsiao came up with an idea of an object-oriented approach to the reliant project, were resources inherit properties from each other. At the same time, the author was working on the USSA parser generator which incorporated object oriented features in multi-language grammar definition. He also came up with an idea of a state change as a result of message passing/receiving between resources. Combination of these ideas has resulted in the current USSA as a core for the new Reliant system. Some latest additions were suggested by Dan Jones.

SPECIAL FLAGS

The following flags should be used only when serious problems occur at USSA run time, like hard to resolve parsing conflicts, or USSA abnormal termination. They suppress certain USSA phases and allow for careful examination of the generated parsing tables.

`d`

If specified, forces USSA to suppress enumeration of states while creating parsing tables. State numbers are assigned only when the verbose description (`'-v'` flag) is printed out.

`f`

If specified, USSA prints out detailed information about lookaheads in the verbose description file (`'-v'` flag).

`w`

If specified, USSA does not split paths leading to the states with reduce/reduce conflicts. Also, `'-d'` flag is set automatically.

`l`

If specified, USSA does not calculate lookaheads. Also, `'-d'` flag is set automatically.

Different combinations of special flags make USSA to produce partially built parsing tables. For example, combination of flags `'-vwl'` prints out in `.v` file the original DFA, without splitting and lookaheads.

`r`

If specified, then USSA generates tokens as local enumerations in the respective C++ class for every USSA type. The values of enumeration are the same as generated constants in .m file (if `'-m'` is specified).

`q`

If specified, then USSA exits after its parsing phase if some declared nodes remain undefined. No C++ sources is generated.

`u`

If specified, then C++ `#line` directives (which refer to the original USSA source file) are not placed before actions in the generated `.C` and `.h` files.

`s`

If specified, optimizes memory use while creating sets of lookahead symbols. Causes slower execution, but may help when memory capacity is limited.

`e ext`

If specified, then `ext` will be used for extension of the outputFile (`C` - default).

`p`

If specified, disables error printout.

`t`

If specified, traces phases of the USSA execution on `stderr`.

`i path`

If specified, the directory `path` will be searched for USSA include files (`/usr/RelX/` by default).

`x`

If specified, reports about nonterminals unreached from start symbol. If any is found, makes USSA exit with nonzero code. Also, sets `'-v'` flag automatically and flags unreached nonterminals in `.v` file.

`o outputFile`

If specified, generated C++ code will be stored in `outputFile`. If not specified, the generated code will be stored in `sourceFile.C` by default.

`v`

If specified, USSA will printout verbose description of the generated minimal LR1 tables. If `'-o'` is specified, description will be placed in `outputFile.v`, otherwise - in `sourceFile.v`.

`h`

If specified, USSA will not create `sourceFile.h` (or `outputFile.h`, if `'-o'` is specified). If not specified, then, depending on the presence or absence of the other options, this file contains `#define` declarations for tokens from the type's rules and respective classes for USSA types. Usually, this flag is not specified.

`m`

If specified, USSA will printout `#define` declarations for tokens from the type's rules in `sourceFile.m` (`outputFile.m` if `'-o'` is specified). This option is used when some, unrelated to USSA, C++ sources need to know the values of tokens in order to communicate with the base monitor. Values defined are the same as the values from the respective local enumerations (see chapter 1.7).

`n`

If specified, USSA will not create parsing tables and definitions of respective C++ classes for types which are defined via `extern` keyword. If other, non-`extern` types, refer to the `extern` types as their base types, then there must be an external action with the C++ `#include` statement for the file containing definitions of the respective classes for the `extern` base types (see next option - `'-b'`). Regardless of `'-n'` option, USSA always generates definitions for non `extern` types and external actions from such types in `.h` file. If `'-n'` is specified, they will also be generated in `.C` file. If `'-n'` is not specified, USSA will generate '`#include "file.h"`' statement in the beginning of the generated `file.C`.

`b`

If specified, USSA will generate parsing tables and respective classes for all defined, even `extern`, types. Parsing tables and actions from rules will be put in `.C` file, definitions of classes in `.h` file. Usually, at least one USSA source file contains `extern` types. This file must be compiled using `'-b'` option. In order to refer to types defined in such a file, other USSA sources must include this file and must be compiled with `'-n'` options. They must also have an external action with C++ `#include` statement for the generated `.h` file.

of actions which should be executed at the end of each of reductions until it makes sure that no reduction in the chain causes a syntax error. This delayed execution is done to avoid side effects which such actions may cause (like damaging internal data, or spawning scripts which affect the whole UNIX system). Also, if an error is detected after some reduction, USSA parser restores parsing stack to the state were the chain of reductions has began and discards the message which has caused that error. If an error is detected during a shift, the erroneous message is also discarded and parser remains in the state were the error occurs. Therefore, any series of unexpected messages is discarded at the same state of LR1 tables until the right message appears. User may avoid discarding of messages, or may define its own error recovery procedure by redefining virtual function `errorBMon::UserRecoverBMon()` (see files `skelbmon.[ch]`). Another feature of real time parsing is that USSA parser always executes actions right after it decides that the chain of reductions is safe, without looking for a lookahead.

Synthesized attributes are accessed from actions in the same way it is done in YACC: via `'$'` variables. However, currently, user have no means to change the type of an attribute (see previous chapter 1.7).

When rules from several base types are united in a single set for the derived type, the order of rules and of associativity declarations is preserved within each base type. Rules and declarations from different types are arranged in order of these types in the inheritance list.

ACTIONS. As it has already been mentioned, actions may be placed outside of any rules: within or outside of types (see previous chapter 1.7). Such external actions may contain any C++ definitions and declarations, as well as preprocessor directives.

1.9 USSA COMMAND LINE

Typing USSA without any arguments causes it to printout a help message on the screen explaining its command line format. If arguments are specified, they must obey the format:

```
[-options...] sourceFile [-options...]
```

The USSA source file `sourceFile` have the default extension `.us` (if the extension is not specified, and there is no file named `sourceFile` in the current directory, then USSA will try to open `./sourceFile.us`). Options are explained below:

```
     token a b c;        // define terminals
     word aa bb cc;      // define nonterminals
```

Equal rules within the same type are allowed and cause no problems unless they have actions or message instructions.

REGULAR EXPRESSIONS. They are allowed in the right part of context-free rules. The meta-rules for the regular expressions are:

```
   regularExpr : name *
               | name +
               | < rightPartOfTheRule > *
               | < rightPartOfTheRule > +
               ;
```

'*' means zero or more times, '+' means one or more times. The following are examples of regular expressions in USSA:

```
LIFE : state +; // one or more instances of 'state'

state : < x y > *;  // zero or more instances of
                    // 'x y'

LIFE : < on | off > +; // one or more instances of
                       // 'on' or 'off'

x : < y { cout << "y" } |     // zero or more
      z { cout << "z"; } > *; // instances of 'y'
                              // or 'z' with a
                              // particular action

u : < a < b | c > * | d +> * v;// complex regular
                               // expression
```

CAUTION. Currently USSA does not process correctly the same regular expression included several times from a number of the same base types. Therefore, it is recommended to use regular expressions only in the type which is not a multiple base class for the other types. Also, some innocent looking regular expressions may cause parsing conflicts. These problems will be fixed in the future.

REAL TIME PARSING AND ERROR RECOVERY. Unlike the LR1 parsing algorithm used by YACC, USSA parser makes several precautions due to the real time nature of its environment. If the USSA parser faces a chain of reductions at run time, it postpones execution

1.8 CONTEXT-FREE RULES

The syntax and semantics of context-free rules in USSA are similar to those from YACC. Rather than providing full description, we refer interested reader to YACC's documentation. However, there are some differences which we list below:

All keywords from YACC, except %type, are valid and have the same sense in USSA. However, the leading '%' need not be present, for example: `token` must be written in USSA instead of `%token` in YACC.

All rules must be finished by `';'` (in YACC the trailing `';'` is optional).

Token definitions need not start from a new line. They must finish by `';'`, for example:

```
left x y z;   token a b c d e;
```

There is no distinct sections for tokens, rules, and executable code divided by '%%' lines as in YACC: rules, token definitions and external actions may be intermixed in any order, for example:

```
x : a;  token b;  a : b;
```

Names of nonterminals and terminals may contain letters, digits, and '_', but the first symbol must be a letter.

Start symbol named `LIFE` is introduced automatically in every type definition, as well names `EndOfStream` and `ACCEPT`. All of them have global scope (see previous chapter 1.7). Also, the following rule is created:

```
ACCEPT : LIFE EndOfStream;
```

It is not recommended to use nonterminal `ACCEPT` in the user written rules. Nonterminal `LIFE` can appear only in the left side of any rule. Because `LIFE` is a start symbol, user must construct its rules so that nonterminals in the base and derived types could be reached from `LIFE` by a chain of rules.

Sometimes actions or message sending instructions must refer to a name which is not a member of any rule of the type. In this case user must define such a name using keywords `left`, `right`, `token`, or `nonassoc` for terminals, or `word` for nonterminals, for example:

extern TYPES

Keyword `extern` before a type definition is used to prevent USSA from generating parsing tables for this type unless the special flag `'-b'` is specified on the USSA command line (see chapter 1.9 for USSA command line format):

```
extern type a [ /* some rules */ ];
```

The `extern` keyword combined with the use of `'-b'` flag is used to allow for separate compilation of USSA sources (see chapter 1.9 for details).

abstract TYPES

Keyword `abstract` before a type definition prevents USSA from generating parsing tables for the type. No nodes of abstract types can be created. Keywords `abstract` and `extern` may be combined. Abstract types are used to prevent generation of redundant C++ code for types which serve only as base types:

```
abstract type a [ /* some rules */ ];
```

include STATEMENT

USSA source files may be included in each other much like C++ sources. `include` statement *must not* end with semicolon. The format of the statement is:

```
include "file"     or      include <file>
```

The first format forces USSA to look for the file `file` in the current directory. The second format implies that `file` is in the directory `/usr/reliant/include/`, unless it is redefined by `'-i'` option on the USSA command line (see chapter 1.9). The `include` statement may only appear outside of any action or C++ code, and outside of any rule, token, or node definition.

MACROS IN THE BASE TYPES

When rules from the base type are added to the rules of the derived type, user may change names of some tokens via macros. Macros are specified together with the links of the base type in the format `oldName = newName`:

```
type a(x) : b(x, u=v, w=t) [/* some rules */];
```

Here `x` is a link, but `u=v` and `w=t` are macros. In the type `a` tokens `u` and `w` from the rules of type `b` will be replaced by tokens `v` and `t` respectively:

```
type b                        type a
u : z w;      becomes         v : z t;
```

MESSAGES AND SYNTHESIZED ATRIBUTES

Synthesized attributes are used in actions to access values associated with the grammar tokens (see details about using attributes in the next chapter 1.8). When USSA lexical analyzer accepts a message it also creates its synthesized attribute of class `sAtribus`. This class is defined in `atribus.h` as follows:

```
class   sAtribus
{
public:
        typeProto *Node; // the node which sent this message
        union            // the message itself
        {
                long    Number;
                char   *Word;
        };
/* constructors and some member functions */
};
```

As it is for the file `nodus.h`, the `#include` statement for `atribus.h` is generated by USSA automatically. If the incoming message has a string value for the synthesized attribute, then field `Word` points to that string. Otherwise, field `Number` has a local enumeration code for the message itself (this code may differ from the code for the same message when the message was sent from `Node`: functions `Send()` always translate these codes). In any case, the following public members of `typeProto` hold the last received values of enumeration code and of a synthesized attribute:

```
        brick      LastToken;        // last message received
                                      // by the node
        char      *LastContentScript;// attribute from this
                                      // message (NULL if
                                      // absent)
```

Here `brick` is a macro defined by USSA either as `short` or as `long`.

PLACEMENT OF EXTERNAL ACTIONS

All external actions from outside of types definitions are gathered and placed before the generated C++ code in the beginning of the generated C++ source file. However, the order of external actions from inside of types definitions is preserved in the generated C++ source. The order of the C++ classes generated for USSA types is also preserved. Therefore, C++ compiler may complain about the undefined forward references. This will be fixed in the future and the order of all types and external actions will be preserved in the generated C++ source.

```
////////////////////////////////////////////////////////////
//The next five functions send a message to a node specified
//by the first argument. The second and the third arguments
//(if any) have the same meaning as above.
////////////////////////////////////////////////////////////

        void  Send(typeProto *, long);
        void  Send(typeProto *, char *);
        void  Send(typeProto *);
        void  Send(typeProto *, long, char *);
        void  Send(typeProto *, char *, char *);

////////////////////////////////////////////////////////////
//The next five functions send a message back to the node
//whose message has been received last (LastSender).
////////////////////////////////////////////////////////////

        void  SendBack(long);
        void  SendBack(char *);
        void  SendBack();
        void  SendBack(long, char *);
        void  SendBack(char *, char *);

///////////////////////////////////////////////////////////
//The next five functions send a message to all descendants
//of the current node.
///////////////////////////////////////////////////////////

        void  SendChilds(long);
        void  SendChilds(char *);
        void  SendChilds();
        void  SendChilds(long, char *);
        void  SendBack(char *, char *);

///////////////////////////////////////////////////////////
//The next five functions send a message to all ancestors
//of the current node, i.e. to all nodes which have the
//current node as a descendant.
///////////////////////////////////////////////////////////

        void  SendParents(long);
        void  SendParents(char *);
        void  SendParents();
        void  SendParents(long, char *);
        void  SendParents(char *, char *);
```

Due to limitation in the implementation of virtual base classes in AT&T C++ front-end, it is impossible to cast the address of a descendant `l()` to its exact type (like `(a *)l()`). However, a user may arrange his hierarchy of types such that all types will have the same common base called `basic`, with the virtual member function `basic *GetBasic()`. User must redefine this function for every type to return `this` pointer. At run time, it must be used either as `l()->GetBasic()` for the descendant `l`, or just as `GetBasic()` for the node itself. This technique implies that the values of interest will be defined in the class `basic`. It may be convenient when the exact type of the accessed node is not known statically.

Another possibility is to use the value of the pointer member `It` and cast it to the statically known type of node.The value of `It` is set at run time automatically for every node.

COMMUNICATION BETWEEN NODES

Messages may be sent not only from the message sending instructions, but also from the actions. This may be convenient, for example, when the action must run some C++ code in order to decide which message to send to which node. Class `typeProto` contains public member functions for this purpose:

```
///////////////////////////////////////////////////////////
// The following five functions send messages to the node
// itself by putting a message in front of the message
// queue. All other sending functions place it at the end.
///////////////////////////////////////////////////////////

// send itself a token (= a value of local enumeration)
        void  Send(long);

// send itself a token (= argument)
        void  Send(char *);

// send to itself the last received token
        void  Send();

// send to itself a token (= a value of local enumeration),
// and a string as a synthesized attribute
        void  Send(long, char *);

// send to itself a token (1st argument), and a string (2nd
// argument) as a synthesized attribute
        void  Send(char *, char *);
```

C++ code from outside of any type, or from an external action may use names of tokens as manifest constants. For every token `n` from type `t` USSA generates a preprocessor definition `#define n_t x`, where `x` is the value of local enumeration for `n` in the generated class `t`. These definitions are placed in the generated `.C`, `.h`, and `.m` files (see chapter 1.9 for more details).

LOCAL AND GLOBAL NAMES

Each terminal and nonterminal has either local or global scope. Scope is specified in the terminals definition statements `token`, `left`, `right`, `nonassoc`, or in the nonterminals definition statements `word` via keywords `local` or `global`:

```
token local x y z;        // local terminals
left global u v;          // global left associative terminals
word global a b c d;      // global nonterminals
word e f g;               // local nonterminals
```

The absence of `local` or `global` keywords implies local scope. A nonterminal defined implicitly just by its appearance in a context-free rule has local scope unless it is redefined via the `‘word global’` statement.

The difference between global and local scopes plays role in derived types. Global names from the base type remain the same in the derived type. On the contrary, a local name `n` from the base type `t` is changed to `_t_n` in all the derived types. Global names are used when different types mention the same terminal or nonterminal in their context-free rules (see example in previous chapter 1.6). Local names are used to prevent name clashes among different base types.

Currently, a reference to a local name from an action is safe only when the type is not a base type of any other type. Otherwise, such a reference may cause C++ compile errors.

SOME USEFUL MEMBERS OF typeProto CLASS

Class `typeProto` is a virtual public base class for any node. It is defined in the supplied file `nodus.h`. USSA generates `#include` statement for this file automatically. The following is an extract from `typeProto` definition:

```
public:
       void      *It;          // address of the node itself
        size_t     Size_us_;     // size of the node
        char      *NodeName;     // this node name
        typeProto *LastSender;   // last sender to this node
virtual basic     *GetBasic() { return 0; };
```

USING LINKS, TERMINALS, AND NONTERMINALS IN RULES, ACTIONS, MESSAGES, AND OTHER C++ CODE

Names of links, global terminals, and global nonterminals may be used freely in actions, rules, and message sending instructions (see next chapter for local and global names). USSA creates local enumeration per each type in order to assign local integer values to these names. When such a value is sent to other nodes, the run time translation guarantees that the receiver gets the right value:

```
type a
[
LIFE : z { cout << "\nz=" << z; }            // z in action
       ( z - ^ ) { cout << "\nsent back"; }  // z in message
     ;
];

type b(a l)
[
LIFE : z ( z - l )                    // z and l in message
       { cout << "\nz sent to a descendant"; }
     ;
];

node a A;    // nodes A and B will exchange message 'z'
node b(A) B;
```

Descendant node named `l` may be accessed from actions using private member function `l()` generated automatically by USSA. This function returns a pointer to a C++ object representing the designated node. The return value of `l()` has a C++ type `typeProto *`. Class `typeProto` is a virtual public base class for all the classes which USSA generates for USSA types. It has some data and member functions defined and initialized automatically, such as a name of the node, pointer to parsing tables, etc.:

```
type c(a l)
[
LIFE : z
       ( - l )
       { cout << "\nsend z=" << z << " to " << l()->NodeName
              << " from " << NodeName; }
     ;
];

node c(A) C; // at run time the printout will be
             // 'send z=2 to A from C'
```

1.7 DETAILS ABOUT TYPES AND NODES

DEFAULT CONSTRUCTORS

Nodes defined in the USSA files are created right after Base Monitor (bm) starts, before any message is passed to any node. bm blocks all UNIX signals during that period of time, and allows them only after all constructors for all nodes will have finish. If a type constructor forks another UNIX process, that process must unblock needed signals itself.

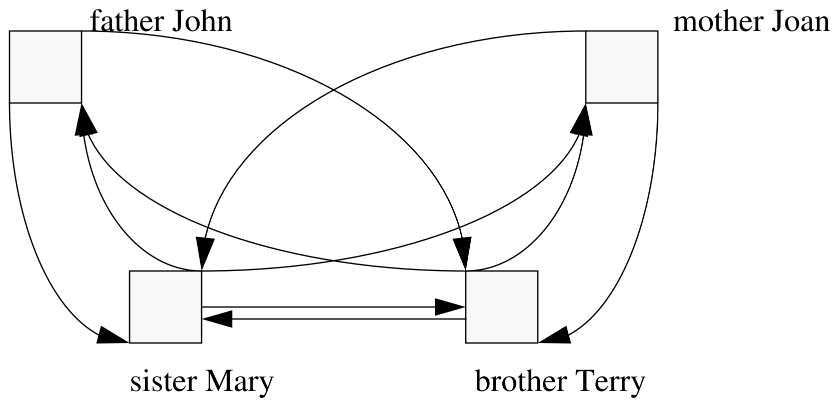
USSA converts a type `t` to a C++ class `t`. For every node `n` of type `t` USSA creates a C++ class named `t_us_n` publicly derived from `t` and a C++ object of this class named `n`. At that time the default constructor for `t` is called (if defined). Therefore, this constructor may run the code common to every node of type `t`. Code for a particular node must be placed in the curly brackets in the node definition:

```
type a
{
public:
 int x;
 a() { /* common initialization code */ }
}
[ /* rules */ ];

node a{ x = 1; } n1,  // run a(), then set x = 1 for node n1
     a{ x = 2; } n2;  // run a(), then set x = 2 for node n2
```

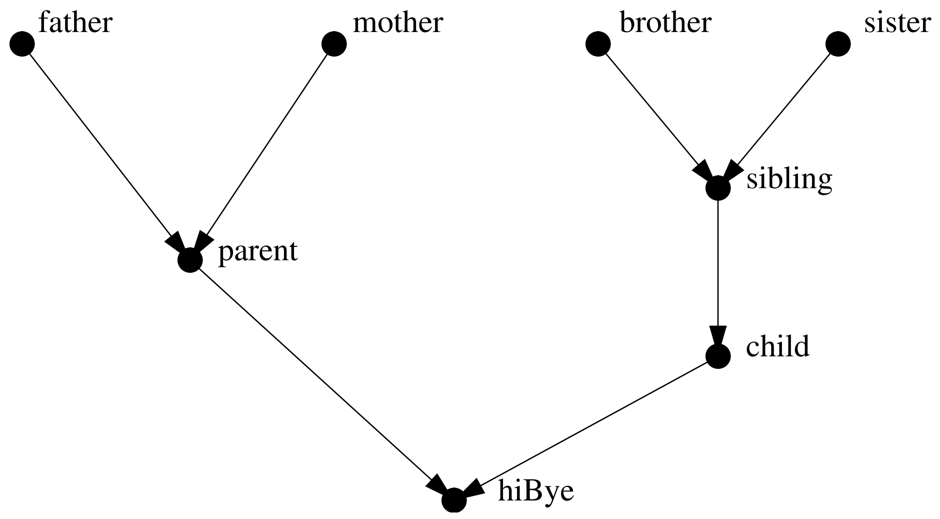
In fact, if a type is defined but there is no node of such a type, the system still creates a global C++ object of the respective class. Therefore, the default constructor, if defined, will be executed at least once on behalf of that object. In order to prevent execution of the critical code within the constructor several times for several nodes, static data members may be used:

```
type a
{
public:
 static int s;
 int x;
 a(){ if ( !s ) { s++; /* execute critical code once */ } }
}
[
 {              // external action:
  int a::s = 0; // initialize static member of the type a
 }
 /* rules */
];
```

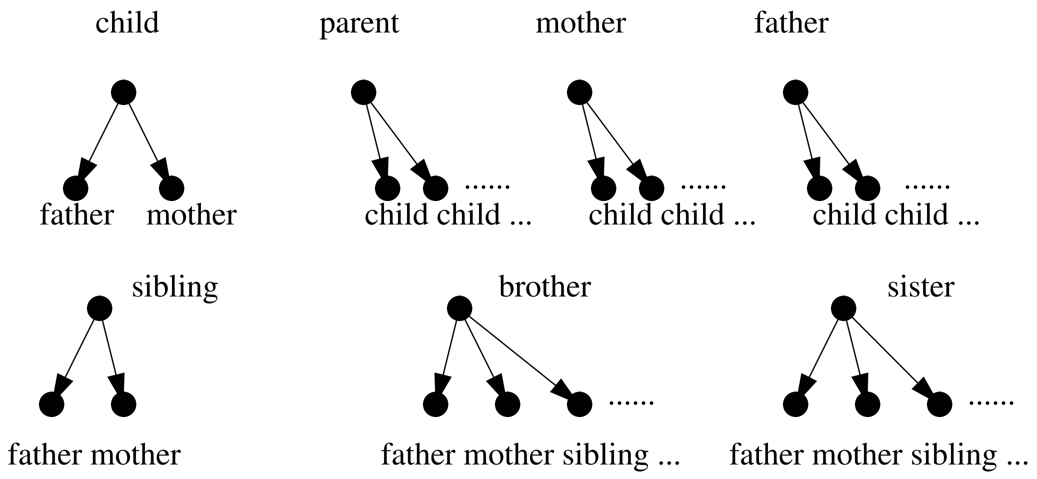
N O D E S



T H E   H I E R A R C H Y   O F T Y P E S



L I N K S   B E T W E E N T Y P E S



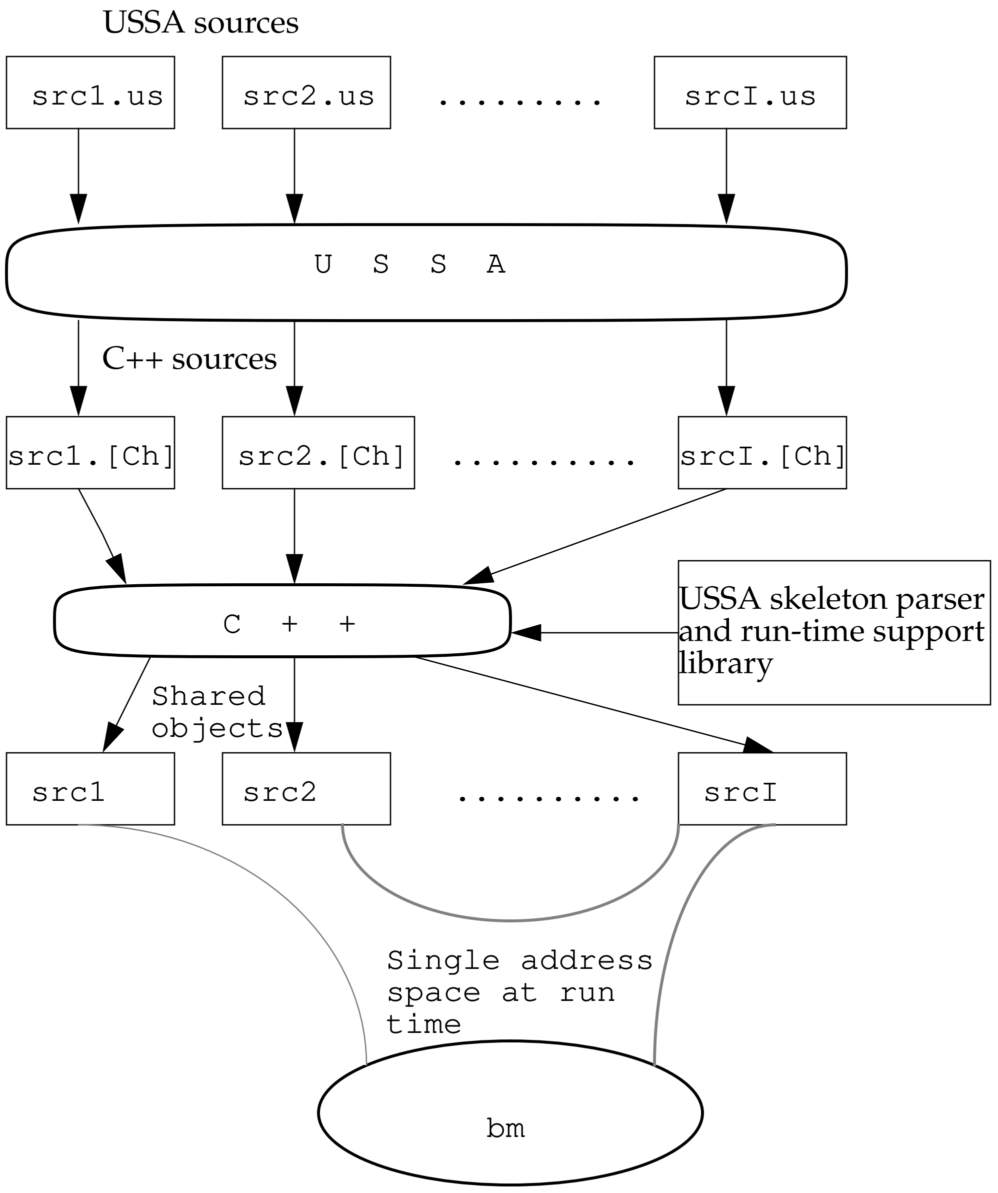
1.6 EXAMPLE

The following example shows definitions of some types and nodes. Graphs show links between nodes, the hierarchy of types, and links for different types:

```
type hiBye { token global Hi Bye; };  // type definition
type parent, mother, father;          // type declaration

type child(father, mother) : public hiBye   //links and
{                                            //inheritance
word global receiveSend;
LIFE            :       receiveSend +;
receiveSend     :       Hi (Bye - >); // receive Hi, send Bye
};

type parent(child ...) : public hiBye // variable number of
{                                     // links
word global sendReceive;
LIFE            :       sendReceive +;
sendReceive     :       (Hi - >) Bye Bye;//send Hi,
};                                          //receive two Bye's

type mother(child ...) : public parent() {};
type father(child ...) : public parent() {};
type sibling(father f, mother m) : public child(f, m)  {};

type brother(father f, mother m, sibling ...) :
                                       public sibling(f, m)
{ receiveSend   :       Bye (Bye - ^,<)
                       { cout << "\nI am a brother\n"; }; };

type sister(father f, mother m, sibling ...) :
                                       public sibling(f, m)
{ receiveSend   :       Bye (Bye - ^,<)
                        { cout << "\nI am a sister\n"; }; };

declare node brother Terry, sister Mary, // node declaration
             mother Joan, father John;

//////////////////////////////////////////////////////////////
// definition of nodes: descendants may have derived, not
// the exact types of links
//////////////////////////////////////////////////////////////
node brother(father John, mother Joan, sibling Mary) Terry,
     sister (father John, mother Joan, sibling Terry) Mary,
     father (child Terry, child Mary)          John,
     mother (child Terry, child Mary)          Joan;
```

object. The Base Monitor provides message passing mechanism and invokes parsers of the appropriate nodes to parse the message flow. The parsers, in turn, invoke actions while reducing context-free rules.

Several shared objects may be prepared from several files with USSA sources. At least one of such shared objects must be linked with bm and USSA libraries statically, and if any two of these sources refer each others types and/or nodes, then at least one of these two must be linked with bm statically (limitations come from UNIX dynamic linker). Linking compiled types and nodes as shared objects allows for the modification of the set of active nodes at run time via UNIX SVR4 `dlopen/dlclose` system calls without restarting the whole system. The following scheme shows the process of preparing shared objects:



1.4 ACTIONS

A piece of C++ code enclosed in curly brackets and stored in the right part of a rule is called action. It is executed when the rule (or its part) is reduced by the parser while parsing the message flow. Action is treated as a protected member function of the type (remember: type is a C++ class + context-free rules). Therefore, it may access other data members of the type and may call its member functions. In the next example the value of `X` will be printed every time `message` is received:

```
type x
{
public:
 int X;
}
[
LIFE : < message { cout << "\nX = " << X; } >*;
];
```

A piece of C++ code in curly brackets outside of any rule in the rule's section or outside types may be used to store additional C++ classes, objects, preprocessor directives, e.t.c. This code is called external action:

```
{ #include <iostream.h> }   // external action
type x
{
public:
 int X;
}
[
{ #include "userFile.h" }   // external action
LIFE : < message { cout << "\nX = " << X << userObject; } >*;
];
```

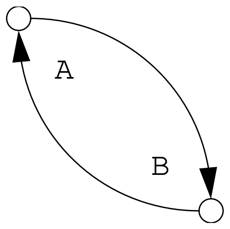
1.5 RUN-TIME ENVIRONMENT.

After definitions of some types and nodes have been written in USSA metalanguage, they must be prepared to run. First, USSA parses these definitions and generates a C++ code. In this code, USSA types will be converted to C++ classes, USSA nodes will be converted to C++ objects, and minimal LR1 parsing tables will be generated from the type's context-free rules - a set of tables per each type. Also, the run-time support data and functions to support communications and actions will be generated. After that, generated sources will be compiled by C++ compiler and then linked with the USSA skeleton parser and run-time support library in a UNIX SVR4 dynamic shared object file. Finally, the Base Monitor (called bm) is executed which dynamically links with the just created shared

```
destination : name    // node or descendant called name
            | empty   // send to all descendants
            | ^       // send to the last sender
            | .       // send to itself
            | <       // send to all ancestors
            | >       // send to all descendants
```

When nodes are linked in a cyclic structure, declarations of types and nodes must appear prior to their definitions. This is because neither of them could be defined without information about the others. Declaration of a type is indicated by the keyword `type` followed by it's name with no members and/or rules. Declaration of a node is designated by keywords `declare node` followed by the node name:

```
type a;                      // declaration of type a
type b(a x) [ /* ... */ ];  // definition of type b
type a(b x) [ /* ... */ ];  // definition of type a

declare node A;              // declaration of node A
node b(A) B;                 // definition of node B
node a(B) A;                 // definition of node A
```

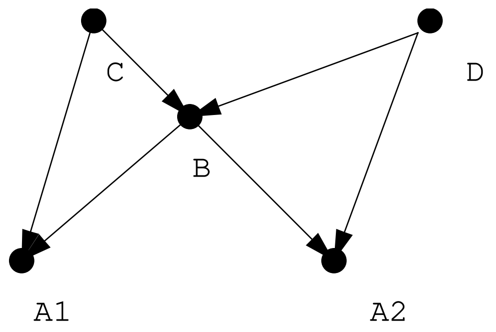


Derived types inherit all links from their base types. They must specify a list of descendants which includes all descendants from the base types. Names of links may differ from the names in the base type definition; common names for links in inherited and base types denote the same link:

```
type a [ /* ... */ ];
type b(a x, a y) [ /* ... */ ]; // two links of type a
type c(a x, b y) [ /* ... */ ]; // links of types a and b
type d(a x, b z) : c(x, z), b(x, x), a [ /* ... */ ];
```

In the above example `d` is derived from `c`, `b`, and `a`. It has two links `x` and `z`. These links are also specified in `c` and `b`. The following graph consists of some nodes of the above defined types:

```
node a A1;
node a A2;
node b(A1, A2) B;
node c(A1, B) C;
node d(A2, B) D;
```



Messages may be sent not only to a particular descendant, but to any other nodes: their names must be specified in the message sending instruction, like

```
        (message - nodeName).
```

The general format of message sending instruction is:

```
( message <,message>* - destination <,destination>* )

// above means 'send all specified messages
// to all specified destinations'

message : empty      // last accepted message
        | string     // terminal or nonterminal named string
        | 'string'   // literal 'string'
```

```
node b node1; // 1st node
node b node2; // 2nd node
```

An optional C++ initialization code is stored in curly brackets after the type name. This code is executed at the time the node is created:

```
node b { x = 1;} node3;
```

Currently, nested types and nodes inside types are not allowed.

1.3 COMMUNICATION BETWEEN NODES.

Nodes accept messages from each other at run time. The USSA parser uses tables associated with the node to parse these messages according to rules from the node's type. Some rules specify what messages will be sent to which nodes. The message sending instruction is placed in the right part of the rule in parens. The following rule sends the message `reply` to the node whose message has been accepted last:

```
type a [ gotIt : message ( reply - ^ ); ];
```

Some nodes may be linked to the other nodes so that the entire set of nodes becomes a directed (may be with cycles) graph. Links must be specified in the type definition as parametrized names of linked nodes and are used in the rules as the destination addresses. When a node of a type with links is created, they are substituted with the names of known nodes. The node itself is called ancestor, and its linked nodes are called descendants. Descendants may have the exact type as their parameters, or may be derived from them. They are indicated as a list in parens following the type name in the type and node definitions:

```
type typeName( type1 n1, type2 n2, . . . ) {/**/} [/**/];
node typeName( type1 n1, type2 n2, . . . ) nodeName;
```

Here `typeI` is the name of type, `nI` is the name of a link (descendant). The last `nI` in the type definition may be indicated as `'...'`, which means there could be any (0 or more) numbers of descendants of `typeI`. In the node definition `typeI` may be omitted. In the type definition `nI` (not equal to `'...'`) may be omitted if it is not used in member functions or syntax rules. The following example shows a single link:

```
type d [ /* some rules */ ];
type a(d x) //////////////////////// descendant x of type d
[ gotIt : message ( reply - x ); // send 'reply' to node x ];

node d node1; /////////node1 of type d
node a(node1) node2; //node2 of type a sends 'reply' to node1
```

Here, `'|'` means 'or', `'*'` - repetition. As in context-free grammars, symbols `opened, closed,` and `i_o_in_progress` are nonterminals, but `OpenFunction, CloseFunction, ReadFunction,` and `WriteFunction` - terminals.

In USSA, types and nodes are used to model an application in the way classes and objects are used in C++. Each node must be of a particular type. A type defines a C++ class and a number of context-free rules which control communication between nodes. Nodes may be linked to allow for convinient and optimal communication. Context-free rules are applied in order to parse the flow of incoming messages. A rule may have embedded C++ code (called action) which is executed when the rule is applied.

Chapters 1.2 -1.6 contain fast presentation of USSA. Details start from chapter 1.7.

1.2 TYPES AND NODES

A type in USSA is a C++ class annotated with a number of context free rules. Rules are written in square brackets after the class definition. The keyword `type` is used instead of the C++ keyword `class`. For example:

```
type a
{ ///////////////////////// data and member functions
public:
 int x;
}
[ /////////////////////// rules
 word global message; // 'message' is a nonterminal
 LIFE : message*;     // 'LIFE' is a sequence of 'message's
];
```

Data and member functions are optional. Types inherit data and member functions from other types (as C++ classes do in v3.0). Rules from the base types are always added to the derived type:

```
type b : public a
[ message : START | STOP; ];
```

Derived type `b` has two rules: the one defined and the one inherited from type `a`:

```
LIFE : message*; ///////////////// inherited rule
message : START | STOP; ///////// defined rule
```

A node is an instance of a type. It has data and member functions associated with the class portion from the type definition. Rules from type definition are also associated with the node. In fact, the minimal LR1 parsing tables generated automatically by USSA from the rules are used in order to parse a flow of messages coming to the node. Different nodes of the same type share a single copy of tables. A node is defined via keyword `node` followed by the type name and then the node name:

USSA

1.0 ABSTRACT

The metalanguage of USSA (Universal Syntax and Semantics Analyzer) specifies software systems in the declarative style using object oriented methodology. The system in question is presented as a collection of objects each annotated with a number of context-free rules which control receiving and passing of messages. It works under UNIX SVR4 and C++ v3.0.

Note: the following text implies some knowledge of the object-oriented design and of C++ language. Also, an elementary notion of context-free grammars is expected.

1.1 INTRODUCTION. USSA -VARIOUS POINTS OF VIEW

There are several equivalent intuitive ideas behind the USSA metalanguage. We present two of them: 'objects with states' and 'objects with rules'.

Objects with states

During the lifetime of an object it undergoes a number of state changes as a result of applying various methods. For example: a C++ file stream becomes 'opened' after successful completion of the open() function. Then, some i/o operations may be in progress.Eventually it becomes 'closed' upon calling the close() function. Therefore, its sequence of states and applied methods may be written as:

```
opened : closed OpenFunction;
closed : opened i_o_in_progress CloseFunction;
```

In USSA, each object is annotated with a number of rules describing its states and the order of transition between different states as a reaction to various methods. In the above examples `opened, closed`, and `i_o_in_progress` are states, but `OpenFunction` and `CloseFunction` are methods. Methods are viewed as messages which must be accepted in the order specified by the rules.

Objects with rules

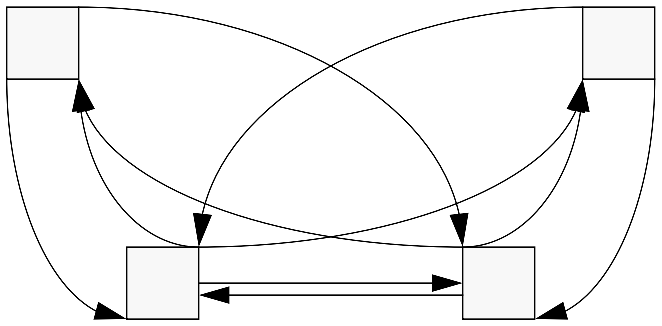
Each object understands certain messages and sequences of messages. Messages which it does not understand are ignored, as well as those coming in the wrong order. For example: an attempt to read a closed file is ignored. The flow of messages is viewed as a sentence from some language. Therefore, an object is annotated with context-free rules describing that language. The following rules describe such a language for C++ file streams:

```
opened          : closed OpenFunction;
closed          : opened i_o_in_progress CloseFunction;
i_o_in_progress : <ReadFunction | WriteFunction>*;
```

USSA

Boris Burshteyn, Pyramid Technology, San Jose CA, bburshte@pyramid.com

Practical guide



CONTENTS