GenoM Manual

For GenoM version 3

Sara Fleury – sara.fleury@laas.fr
Matthieu Herrb – matthieu.herrb@laas.fr
Anthony Mallet – anthony.mallet@laas.fr
Cédric Pasteur

July 6, 2010

Contents

1

Introduction

2

Component model

3

G^{en}oM overview

4

A minimal example

5

Input file format

This chapter describes the G^en^oM Input File Format (dotgen) semantics and gives the syntax for dotgen grammatical constructs.

5.1 Overview

The G^en^oM Input File Format (dotgen) is the language used to formally describe a G^en^oM component in terms of services and data types it provides. A description written in dotgen completely defines the interface and the internals of a component.

A description of the dotgen preprocessing is presented in section 5.2, Preprocessing. The grammar is presented in section 5.3, Dotgen grammar, and associated semantics is described in the rest of this chapter either in place or through references to other sub sections of this chapter.

A source file containing a dotgen component specification must have a ".gen" extension. The description of the dotgen grammar uses a syntax notation that is similar to Extended Backus-Naur Format (EBNF). Table 5.1, dotgen EBNF symbols, lists the symbols used in this format and their meaning.

5.2 Preprocessing

A dotgen specification consists of one or more files that are preprocessed. The preprocessing performs file inclusion and macro substitution and is controlled by directives introduced by lines having # as the first character other than white space. The preprocessing is done by the C preprocessor available on the host system and configured during the build of G^en^oM. It is invoked as a separate process.

Symbol	Meaning
::=	Definition.
|	Alternation.
text	Nonterminals.
"text"	Terminals.
(...)	Grouping.
{ ... }	Repetition: may occur zero or any number of times.
[...]	Option: may occur zero or one time.

Table 5.1: dotgen EBNF symbols

Preprocessor directives beginning with # have their own syntax (namely, the C preprocessor syntax), independent of the dotgen language and not described in this document. See for instance [1] for a documentation. Directives may appear anywhere in the source file but are not seen nor interpreted by G^{en}oM. The primary use of the preprocessing facilities is to include definitions (especially type definitions) from other dotgen specifications. Text in files included with a `#include` directive is treated as if it appeared in the including file.

5.3 Dotgen grammar

```
(1)   spec                ::= { statement } statement
(2)   statement           ::= idlstatement
                            | genomstatement
                            | cpphash
(3)   idlstatement        ::= ( module | const_dcl | type_dcl ) ";"
(4)   genomstatement      ::= ( component | attribute | port | task | service ) ";"
(5)   component           ::= "component" identifier [ "{" attr_list "}" ]
(6)   port                ::= port_dir "<" port_type ">" identifier
(7)   port_type           ::= [ type_spec ]
(8)   port_dir            ::= "inport" "data"
                            | "inport" "event"
                            | "outport" "data"
                            | "outport" "event"
(9)   attribute           ::= "attribute" nodir_param_list
(10)  task                ::= "task" identifier [ "{" attr_list "}" ]
(11)  service             ::= "service" identifier "(" param_list ")" "{" attr_list
                              "}"
(12)  attr_list           ::= { attr ";" } attr ";"
(13)  attr                ::= "doc" ":" string_literals
                            | "ids" ":" named_type
                            | "version" ":" string_literals
                            | "lang" ":" string_literals
                            | "email" ":" string_literals
                            | "require" ":" string_list
                            | "build-require" ":" string_list
                            | "clock-rate" ":" const_expr time_unit
                            | "period" ":" const_expr time_unit
                            | "delay" ":" const_expr time_unit
                            | "priority" ":" positive_int_const
                            | "stack" ":" positive_int_const size_unit
                            | "throw" ":" event_list
                            | "task" ":" identifier
                            | "interrupts" ":" identifier_list
                            | "before" ":" identifier_list
                            | "after" ":" identifier_list
                            | "validate" ":" validate
                            | "codel" codel
(14)  validate            ::= identifier "(" param_list ")"
(15)  codel               ::= event_list ":" identifier "(" param_list ")" "yield"
                              event_list
(16)  event_list          ::= identifier_list
```

```
(17)  param_list          ::= [ { param "," } param ]
(18)  nodir_param_list    ::= { nodir_param "," } nodir_param
(19)  param               ::= param_dir named_param [ "=" initializer ]
(20)  nodir_param         ::= named_param [ "=" initializer ]
(21)  named_param         ::= param_member
                            | param_member "::" identifier
                            | "::" identifier
(22)  param_member        ::= identifier
                            | param_member "." identifier
                            | param_member "[" positive_int_const "]"
(23)  param_dir           ::= "in"
                            | "out"
                            | "inout"
                            | "inport"
                            | "outport"
(24)  initializer_list    ::= [ { initializer "," } initializer ]
(25)  initializer         ::= initializer_value
                            | "::" string_literals
                            | initializer_value "::" string_literals
(26)  initializer_value   ::= const_expr
                            | "{" initializer_list "}"
                            | "[" positive_int_const "]" "=" const_expr
                            | "[" positive_int_const "]" "=" "{" initializer_list
                              "}"
                            | "." identifier "=" const_expr
                            | "." identifier "=" "{" initializer_list "}"
(27)  module              ::= "module" module_name "{" idlspec "}"
(28)  module_name         ::= identifier
(29)  idldef              ::= [ idlstatement | cpphash ]
(30)  idlspec             ::= { idldef } idldef
(31)  const_dcl           ::= "const" const_type identifier "=" const_expr
(32)  const_type          ::= integer_type
                            | char_type
                            | boolean_type
                            | floating_pt_type
                            | octet_type
                            | string_type
                            | named_type
(33)  type_dcl            ::= constructed_type
                            | "typedef" alias_list
                            | forward_dcl
(34)  constructed_type    ::= struct_type
                            | union_type
                            | enum_type
(35)  alias_list          ::= ( type_spec | alias_list "," ) declarator
(36)  struct_type         ::= "struct" scope_push_struct "{" member_list "}"
(37)  union_type          ::= "union" scope_push_union "switch" "("
                              switch_type_spec ")" "{" switch_body "}"
(38)  enum_type           ::= "enum" identifier "{" enumerator_list "}"
(39)  forward_dcl         ::= ( "struct" | "union" ) identifier
```

```
(40)  declarator            ::= simple_declarator
                              | array_declarator
(41)  simple_declarator     ::= identifier
(42)  array_declarator      ::= ( simple_declarator | array_declarator )
                                fixed_array_size
(43)  fixed_array_size      ::= "[" positive_int_const "]"
(44)  type_spec             ::= simple_type_spec
                              | constructed_type_spec
(45)  simple_type_spec      ::= base_type_spec
                              | template_type_spec
                              | named_type
(46)  constructed_type_spec ::= constructed_type
(47)  named_type            ::= scoped_name
(48)  base_type_spec        ::= boolean_type
                              | integer_type
                              | floating_pt_type
                              | char_type
                              | octet_type
                              | any_type
(49)  template_type_spec    ::= sequence_type
                              | string_type
                              | fixed_type
(50)  integer_type          ::= signed_int
                              | unsigned_int
(51)  floating_pt_type      ::= float_type
                              | double_type
(52)  signed_int            ::= signed_longlong_int
                              | signed_long_int
                              | signed_short_int
(53)  unsigned_int          ::= unsigned_longlong_int
                              | unsigned_long_int
                              | unsigned_short_int
(54)  unsigned_short_int    ::= "unsigned" "short"
(55)  unsigned_long_int     ::= "unsigned" "long"
(56)  unsigned_longlong_int ::= "unsigned" "long" "long"
(57)  signed_short_int      ::= "short"
(58)  signed_long_int       ::= "long"
(59)  signed_longlong_int   ::= "long" "long"
(60)  float_type            ::= "float"
(61)  double_type           ::= "double"
(62)  char_type             ::= "char"
(63)  boolean_type          ::= "boolean"
(64)  octet_type            ::= "octet"
(65)  any_type              ::= "any"
(66)  string_type           ::= "string" [ "<" positive_int_const ">" ]
(67)  sequence_type         ::= "sequence" "<" simple_type_spec ( ","
                                positive_int_const ">" | ">" )
```

```
(68)  fixed_type            ::= "fixed" [ "<" positive_int_const ","
                                  positive_int_const ">" ]
(69)  switch_type_spec      ::= integer_type
                              | char_type
                              | boolean_type
                              | enum_type
                              | named_type
(70)  switch_body           ::= { case } case
(71)  member_list           ::= { member ";" } member ";"
(72)  member                ::= ( type_spec | member "," ) declarator
(73)  case                  ::= case_label_list type_spec declarator ";"
(74)  case_label_list       ::= { case_label } case_label
(75)  case_label            ::= ( "case" const_expr | "default" ) ":"
(76)  enumerator_list       ::= { enumerator "," } enumerator
(77)  enumerator            ::= identifier
(78)  scope_push_struct     ::= identifier
(79)  scope_push_union      ::= identifier
(80)  scoped_name           ::= [ [ scoped_name ] "::" ] identifier
(81)  const_expr            ::= or_expr
(82)  positive_int_const    ::= const_expr
(83)  or_expr               ::= { xor_expr "|" } xor_expr
(84)  xor_expr              ::= { and_expr "^" } and_expr
(85)  and_expr              ::= { shift_expr "&" } shift_expr
(86)  shift_expr            ::= { add_expr ( ">>" | "<<" ) } add_expr
(87)  add_expr              ::= { mult_expr ( "+" | "-" ) } mult_expr
(88)  mult_expr             ::= { unary_expr ( "*" | "/" | "%" ) } unary_expr
(89)  unary_expr            ::= [ "-" | "+" | "~" ] primary_expr
(90)  primary_expr          ::= literal
                              | "(" const_expr ")"
                              | named_type
(91)  literal               ::= "TRUE"
                              | "FALSE"
                              | integer_literal
                              | "<float_literal>"
                              | "<fixed_literal>"
                              | "<char_literal>"
                              | string_literals
(92)  string_literals       ::= { string_literal } string_literal
(93)  string_list           ::= { string_literals "," } string_literals
(94)  time_unit             ::= [ "s" | "ms" | "us" ]
(95)  size_unit             ::= [ "k" | "m" ]
(96)  identifier            ::= "[A-Za-z_][A-Za-z0-9_]*"
                              | "s"
                              | "ms"
                              | "us"
                              | "k"
                              | "m"
                              | "component"
```

```
                                | "ids"
                                | "attribute"
                                | "version"
                                | "lang"
                                | "email"
                                | "require"
                                | "build-require"
                                | "clock-rate"
                                | "task"
                                | "period"
                                | "delay"
                                | "priority"
                                | "stack"
                                | "codel"
                                | "validate"
                                | "yield"
                                | "throw"
                                | "doc"
                                | "interrupts"
                                | "before"
                                | "after"
                                | "event"
                                | "data"
                                | "inport"
                                | "outport"
                                | "in"
                                | "out"
                                | "inout"
(97)    identifier_list         ::= { identifier "," } identifier
(98)    cpphash                 ::= "#" integer_literal string_literal ( "\n" |
                                    integer_literal "\n" )
```

5.4 Dotgen specification

A dotgen specification consists of one or more statements. Statements are either IDL statements, G[en]oM statements or cpp line directives. The syntax is:

```
(1)     spec                    ::= { statement } statement
(2)     statement               ::= idlstatement
                                  | genomstatement
                                  | cpphash
(3)     idlstatement            ::= ( module | const_dcl | type_dcl ) ";"
(4)     genomstatement          ::= ( component | attribute | port | task | service ) ";"
```

Definitions are named be the mean of identifiers: see section 5.5, Identifiers and reserved keywords. Cpp line directives are normally issued by the C preprocessor and are used to define the current input file name and line number (section 5.6, Line directives).

An IDL statement defines types (see section 5.8, Type declaration), constants (see section 5.9, Constant declaration) or IDL modules containing types and constants (see section 5.7, Module declaration). The syntax follows closely the subset the OMG IDL specification corresponding to type and constants definitions (see Part I, chapter 7 of [2]). Note that this subset of the dogten grammar is not in any manner tied to OMG IDL and may diverge from future OMG specifications.

A G[en]oM statement defines components, communication ports, services and execution contexts called tasks.

5.5 Identifiers and reserved keywords

An identifier is a sequence of ASCII alphabetic, digit, and underscore ("_") characters. The first character must be an ASCII alphabetic character.

```
(96)  identifier          ::= "[A-Za-z_][A-Za-z0-9_]*"
                            | "s"
                            | "ms"
                            | "us"
                            | "k"
                            | "m"
                            | "component"
                            | "ids"
                            | "attribute"
                            | "version"
                            | "lang"
                            | "email"
                            | "require"
                            | "build-require"
                            | "clock-rate"
                            | "task"
                            | "period"
                            | "delay"
                            | "priority"
                            | "stack"
                            | "codel"
                            | "validate"
                            | "yield"
                            | "throw"
                            | "doc"
                            | "interrupts"
                            | "before"
                            | "after"
                            | "event"
                            | "data"
                            | "inport"
                            | "outport"
                            | "in"
                            | "out"
                            | "inout"
```

Words that are reserved keywords in the dotgen language are valid identifiers where their use is not ambiguous.

5.6 Line directives

Line directives are normally not used. They are inserted by `cpp`, as a result of preprocessing the input file (section 5.2). They can be used mostly to achieve special effects on error reporting or similar.

A line directive starts with the # sign:

```
(98)  cpphash            ::= "#" integer_literal string_literal ( "\n" |
                             integer_literal "\n" )
```

The # sign is followed by the current line number and file name of the source file, optionally followed by a numeric flag. The flag is never used by G^{en}oM. Its meaning depends on the C preprocessor used: see for instance chapter 1.8, C preprocessor output, of the FSF C preprocessor [1].

The file name and line number replace the current value kept internally by G^{en}oM and are used in error reporting messages as well as a few other places.

5.7 Module declaration

A module definition satisfies the following syntax:

```
(27)  module             ::= "module" module_name "{" idlspec "}"
(28)  module_name        ::= identifier
(29)  idldef             ::= [ idlstatement | cpphash ]
(30)  idlspec            ::= { idldef } idldef
```

The only effect of a module is to scope IDL identifiers. It is similar to a C++ or Java namespace; it is considered good practice to enclose your type definitions inside a module definition to prevent name clashes between components.

5.8 Type declaration

Type declarations define new data types and associate a name (an identifier) with it. The `typedef` keyword can be used to name an existing type. The constructed types `struct`, `union` and `enum` also name the type they define. The syntax is the following:

```
(33)  type_dcl           ::= constructed_type
                           | "typedef" alias_list
                           | forward_dcl
(34)  constructed_type   ::= struct_type
                           | union_type
                           | enum_type
(35)  alias_list         ::= ( type_spec | alias_list "," ) declarator
```

A type specification is the description of a type. It can be used in a `typedef` construct or anywhere a typed value is expected.

```
(44)  type_spec          ::= simple_type_spec
                           | constructed_type_spec
(45)  simple_type_spec   ::= base_type_spec
                           | template_type_spec
                           | named_type
(48)  base_type_spec     ::= boolean_type
                           | integer_type
                           | floating_pt_type
                           | char_type
                           | octet_type
                           | any_type
```

```
(49)  template_type_spec    ::= sequence_type
                              | string_type
                              | fixed_type
(46)  constructed_type_spec ::= constructed_type
(47)  named_type            ::= scoped_name
(80)  scoped_name           ::= [ [ scoped_name ] "::" ] identifier
(40)  declarator            ::= simple_declarator
                              | array_declarator
(41)  simple_declarator     ::= identifier
```

5.9 Constant declaration

```
(31)  const_dcl             ::= "const" const_type identifier "=" const_expr
```

6

G^enoM IDL mappings

G^enoM IDL is independent of the programming language used to implement the services and internals of a component. In order to use the G^enoM generated source code, it is necessary for programmers to know how to access the service parameters and ports from their programming languages. This chapter defines the mapping of G^enoM IDL constructs to the supported programming languages.

The mapping between G^enoM IDL and a programming language diverges from the OMG CORBA standard. This is unfortunate, because this might lead to some confusion for the developers used to OMG CORBA, but it was necessary to define mappings well targetting real-time platforms. The design strategy that guided the definition of those mappings was to try to have contiguous memory segments, that do not require memory management primitives, for most of the data types. Only unbounded string and sequences do not follow this scheme.

G^enoM currently implements mappings for the `C` and `C++` languages. For the `C` language, see section 6.1, C mappings. For the `C++` language, see section 6.2, C++ mappings,

6.1 C mappings

6.1.1 Scoped names

The `C` mappings always use the global name for a type or a constant. The `C` global name corresponding to a G^enoM IDL global name is derived by converting occurrences of "::" to "_" (an underscore) and eliminating the leading underscore.

6.1.2 Mapping for constants

In `C`, constants defined in dotgen are `#define`d. For instance, the following IDL:

```
const long longint = 1;
const string str = "string example";
```

would map into

```
#define longint 1
#define str "string example"
```

The identifier can be referenced at any point in the user's code where a literal of that type is legal.

IDL	C
boolean	bool
unsigned short	uint16_t
short	int16_t
unsigned long	uint32_t
long	int32_t
unsigned long long	uint64_t
long long	int64_t
float	float
double	double
char	char
octet	uint8_t
any	type any not implemented yet

Table 6.1: Basic data types C mappings

6.1.3 Mapping for basic data types

The basic data types have the mappings shown in Table 6.1, Basic data types C mappings. Integer types use the `C99` fixed size integer types as provided by the `stdint.h` standard header. Users do not have to include this header: the template mapping generation procedure output the appropriate `#include` directive along with the mappings for the integer types.

6.1.4 Mapping for enumerated types

The C mapping of an IDL `enum` type is an unsigned, 32 bits wide integer. Each enumerator in an enum is `#defined` with an appropriate unsigned integer value conforming to the ordering constraints.

For instance, the following IDL:

```
enum e {
   value1,
   value2
};
```

would map, according to the scoped names rules, into

```
typedef uint32_t e
#define e_value1  1
#define e_value2  2
```

6.1.5 Mapping for strings

G^enoM IDL bounded strings are mapped to nul terminated character arrays (i.e., C strings). Unbounded strings are mapped to a pointer on such a character array.

For instance, the following OMG IDL declarations:

```
typedef string unbounded;
typedef string<16> bounded;
```

would map into

```
typedef char *unbounded;
typedef char bounded[16];
```

6.1.6 Mapping for arrays

G^enoM IDL arrays map directly to C arrays. All array indices run from 0 to `size-1`.

For instance, the following IDL:

```
typedef long array[4][16];
```

would map into

```
typedef int32_t array[4][16];
```

6.1.7 Mapping for structure types

G^enoM IDL structures map directly onto C `structs`. Note that these structures may potentially include padding.

For instance, the following IDL:

```
struct s {
   long a;
   long b;
};
```

would map into

```
typedef struct {
   int32_t a;
   int32_t b;
} s;
```

6.1.8 Mapping for union types

G^enoM IDL unions map onto C `structs`. The discriminator in the enum is referred to as `_d`, the union itself is referred to as `_u`.

For instance, the following IDL:

```
union u switch(long) {
   case 1: long a;
   case 2: float b;
   default: char c;
};
```

would map into

```
typedef struct {
   int32_t _d;
   union {
      int32_t a;
```

```
        float b;
        char c;
    } _u;
} u;
```

6.1.9 Mapping for sequence types

G^en oM IDL sequences mapping differ slightly for bounded or unbouded variations of the sequence. Both types maps onto a C `struct`, with a `_maximum`, `_length` and `_buffer` members.

For unbounded sequences, `buffer` points to a buffer of at most `_maximum` elements and containing `_length` valid elements. An additional member `_release` is a function pointer that can be used to release the storage associated to the `_buffer` and reallocate it. It is the responsibility of the user to maintain the consistency between those members.

For bounded sequences, `buffer` is an array of at most `_maximum` elements and containing `_length` valid elements. Since `_buffer` is an array, no memory management is necessary for this data type.

For instance, the following IDL:

```
typedef sequence<long> unbounded;
typedef sequence<long,16> bounded;
```

would map into

```
typedef struct {
   uint32_t _maximum, _length;
   int32_t *_buffer;
   void (*release)(void *_buffer);
} unbounded;

typedef struct {
   const uint32_t _maximum;
   uint32_t _length;
   int32_t _buffer[16];
} bounded;
```

6.2 C++ mappings

6.2.1 Scoped names

The `C++` mappings for scoped names use C++ scopes. IDL `module`s are mapped to `namespace`s. For instance, the following IDL:

```
module m {
   const string str = "scoped string";
};
```

would map into

```
namespace m {
   const std::string str = "scoped string";
}
```

IDL	C++
boolean	bool
unsigned short	uint16_t
short	int16_t
unsigned long	uint32_t
long	int32_t
unsigned long long	uint64_t
long long	int64_t
float	float
double	double
char	char
octet	uint8_t
any	type any not implemented yet

Table 6.2: Basic data types C++ mappings

6.2.2 Mapping for constants

G[en]oM IDL constants are mapped to a C++ constant. For instance, the following IDL:

```
const long longint = 1;
const string str = "string example";
```

would map into

```
const int32_t longint = 1;
const std::string str = "string example";
```

6.2.3 Mapping for basic data types

The basic data types have the mappings shown in Table 6.2, Basic data types C++ mappings. Integer types use the `C99` fixed size integer types as provided by the `stdint.h` standard header (since the C++ `cstdint` header is not part of the C++ at the time of writing this document). Users do not have to include this header: the template mapping generation procedure output the appropriate `#include` directive along with the mappings for the integer types.

6.2.4 Mapping for enumerated types

The C++ mapping of an IDL `enum` type is the corresponding C++ `enum`. An additional constant is generated to guarantee that the type occupies a 32 bits wide integer.

For instance, the following IDL:

```
enum e {
   value1,
   value2
};
```

would map, according to the scoped names rules, into

```
enum e {
   value 1,
   value 2,
   _unused = 0xffffffff,
};
```

6.2.5 Mapping for strings

G^en oM IDL bounded strings are mapped to nul terminated character arrays (i.e., C strings) wrapped inside the specific `genom::bounded_string` class. Unbounded strings are mapped to `std:string` provided by the C++ standard.

For instance, the following OMG IDL declarations:

```
typedef string unbounded;
typedef string<16> bounded;
```

would map into

```
typedef std::string unbounded;
typedef genom::bounded_string<16> bounded;
```

The `genom::bounded_string` provides the following interface:

```
namespace genom3 {
   template<std::size_t L> struct bounded_string {
      char c[L];
   };
}
```

This minimalistic definition will be refined before the official 3.0 G^en oM release.

6.2.6 Mapping for arrays

G^en oM IDL arrays map directly to C++ arrays. All array indices run from 0 to `size-1`.

For instance, the following IDL:

```
typedef long array[4][16];
```

would map into

```
typedef int32_t array[4][16];
```

6.2.7 Mapping for structure types

G^en oM IDL structures map directly onto C++ `structs`. Note that these structures may potentially include padding.

For instance, the following IDL:

```
struct s {
   long a;
   long b;
};
```

would map into

```
struct s {
   int32_t a;
   int32_t b;
};
```

6.2.8 Mapping for union types

GenoM IDL unions map onto C `structs`. The discriminator in the enum is referred to as `_d`, the union itself is referred to as `_u`.

For instance, the following IDL:

```
union u switch(long) {
   case 1: long a;
   case 2: float b;
   default: char c;
};|
```

would map into

```
struct u {
   int32_t _d;
   union {
      int32_t a;
      float b;
      char c;
   } _u;
};
```

Note that the C++ standard does not allow union members that have a non-trivial constructor. Consequently, the C++ mapping for such kind of unions is not allowed in GenoM either. This concerns `sequences` and `strings`, and structures or unions that contain such a type. You should thus avoid to define such datatypes in GenoM IDL in order to maximize the portability of your definitions.

6.2.9 Mapping for sequence types

GenoM IDL sequences mapping differ for bounded or unbouded variations of the sequence. The unbounded sequence maps onto a C++ `std::vector` provided by the C++ standard. The bounded sequences maps onto the specific `genom3::bounded_vector` class.

For instance, the following IDL:

```
typedef sequence<long> unbounded;
typedef sequence<long,16> bounded;
```

would map into

```
typedef std::vector<int32_t> unbounded;
typedef genom3::bounded_vector<int32_t, 16> bounded;
```

The `genom::bounded_vector` provides the following interface:

```
namespace genom3 {
   template<typename T, std::size_t L> struct bounded_vector {
      T e[L];
   };
}
```

This minimalistic definition will be refined before the official 3.0 G^en oM release.

7

Running G^en^oM

7.1 Synopsis

```
genom3 [-l] [-h] [--version]
genom3 [-I dir] [-D macro[=value]] [-E|-n] [-v] [-d] file.gen
genom3 [general options] template [template options] file.gen
```

7.2 Description

The G^en^oM program is a *source code generator* that is used to generate software components from a formal description file.

The input *file* is expected to contain the description of the services, input and output ports, data types definitions and execution contexts of a software component, written in the *dotgen* language.

The dotgen specification is first processed by a C preprocessor before it is parsed by G^en^oM and transformed into an abstract syntax tree. The program used as a C preprocessor can be changed with the `CPP` environment variable. G^en^oM accepts `-I` and `-D` options that are passed inchanged to the cpp program.

The abstract syntax tree is exported in a format suitable to a *generator engine* that is in charge of a *template* execution for actual source code generation. The generator engine provides a scripting language and a set of procedures for use by templates. The directory where source code for the generator engine is searched can be changed with the `-s` option.

Templates are a set of source files that serve as the basis for source code generation. Templates source files are interpreted by the generator engine. They can contain code written in the scripting language provided by the generator engine, that computes generated output, or regular source code that is appended directly to the generated code. Intermediate files and scripts are saved in a temporary directory before they are copied to the final destination directory. The `-T` option changes the path of the temporary directory. The `-d` option will keep all temporary files instead of deleting them once the program terminates. This is useful only for template development and debugging.

The choice of a template depends on the kind of source code that is wanted by the user. Refer to the documentation of the templates for a description on what they do. The names of the available templates can be listed with the `-l` option. The directory in which templates are looked for can be changed with the `-t` option.

The G^en^oM program accepts *general options* that affect the general program behaviour. G^en^oM can also pass *template options* to the template. These options will only affect the template behaviour.

7.3 General options

-I *dir*

Add the directory *dir* to the list of directories to be searched for included files. The *dir* argument is passed as-is to the `cpp` program via the same `-I` option.

When `-r` option is in effect (either explicitely passed on the command line, or configured by default during the build process), an implicit `-I` directive pointing to the directory of the input file is appended to the end of the list of searched directories.

-D *macro[=value]*

Predefine *macro*, with definition `1` or *value* if given, in the same way as a `#define` directive would do it. This option is passed as-is to the `cpp` program.

If you are invoking genom from the shell, you may have to use the shell quoting character to protect shell's special characters such as spaces.

An implicit macro `__GENOM__` is always defined and contains the version of the genom program. This can be used to divert some lines in source files meant to be included by other tools that genom, and that contain syntax that genom does not understand.

-E

Stop after the preprocessing stage, and do not run genom proper. The output of `cpp` is sent to the standard output. genom exits with a non-zero status if there are any preprocessing errors, such as a non-existent included file.

-n, --parse-only

Stop after the input file parsing stage, and do not invoke any template. This is useful to check the syntax of the input file. Any errors or warning are reported and genom exits with a non-zero status if there are errors.

-l, --list

Print to the standard output the list of available templates. Each line of output contains the name of a template and the genom engine that is uses (currently, only Tcl-based templates are supported).

By default, the standard templates directory is searched, but any `-t` option will be taken into account.

-t *path*, --tmpldir=*path*

Use *path* as the directory containing templates. This can be a colon separated list of directories which are search in order.

This option is useful only for templates not installed in the genom standard directories, i.e. `share/genom/<version>/templates` or `share/genom/site-templates`.

path is searched for files matching `dir/*/template.tcl`, where `*` is the actual template name.

-s *dir*, --sysdir=*dir*

Use *dir* as the directory holding genom engine files. This option is useful if non-standard engines are to be used. The default value is `share/genom/<version>/engines`.

dir should contain directories named after the engine name.

-T *dir*, --tmpdir=*dir*
Use *dir* as the temporary directory holding intermediate files. See also the environment variable `TMPDIR`.

-r, --rename
Some `cpp` programs cannot handle correctly files with a `.gen` extension. This option will make genom call cpp with an input file ending in `.c`, linked to the real input file.

--no-rename
This option is the opposite of `-r`: it forces genom to invoke `cpp` directly with the `.gen` input file. This is the default behaviour. If you are using GNU cpp, it might be necessary to pass it the additional `-xc` option, to force it to handle the file as C source.

-v, --verbose
Force genom to be more verbose while processing input files.

-d, --debug
Activate some debugging options. In particular, temporary files are not deleted. Useful for debugging genom itself or generator engines.

--version
Display the version number of the invoked GenoM.

-h, --help
Print usage summary and exit.

7.4 Template options

-h, --help
Templates might define their own specific options. The `-h` option is always defined, and prints a summary of supported options. See the template manual for a detailed description. Template options should be passed after the template name, and before the input file name.

7.5 Environment variables

CPP
Define the C preprocessor program to use. The default depends on the value configured during the genom build process, but it is most often `gcc -E -xc` on Linux systems.

The `CPP`program must recognize `-I` and `-D` arguments.

GENOM_TMPL_PATH
The value of `GENOM_TEMPLATE_PATH` is a colon-separated list of directories, much like `PATH`, where G^{en}oM looks for templates. Setting this variable overrides the default search path, but any `-t` option takes precedence over this variable.

TMPDIR
Path to the directory holding temporary files. Defaults to `/tmp`.

8

Templates

8.1 The template command

```
template require file
```

Source tcl *file* and make its content available to the template files. The file name can be absolute or relative. If it is relative, it is interpreted as relative to the template directory (see `dotgen template dir`).

Parameters:

file	Tcl input file to source. Any procedure that it creates is made available to the template files.

```
template parse [args list] [file/string/raw file ...]
```

This is the main template function that parses a template source file and instanciate it, writing the result into the current template directory (or in a global variable). This procedure should be invoked for each source file that form a template.
When invoking `template parse`, the last two arguments are the destination file or string. A destination file is specified as `file file` (the filename is relative to the current template output directory). Alternatively, a destination string is specified as `string var`, where *var* is the name of a *global* variable in which the template engine will store the result of the source instantiation.
The output destination file or string is generated by the template from one or several input source. An input source is typically a source file, but it can also be a string or a raw (unprocessed) text. An input source file is specified with `file file`, where *file* is a file name relative to the template directory. An input source read from a string is specified as `string text`, where text is any string, processed by the template engine as usual. Finally, a raw, unprocessed source that is copied verbatim to the destination, is specified as `raw text`, where *text* is any string.
Additionnaly, each input source, defined as above, can be passed a list of optional arguments by using the special `args list` construction as the first argument of the `template parse` command. The list given after `args` can be retrieved from the template source file from the usual `argv` variable.

Examples:

```
template parse file mysrc file mydst
```

Will parse the input file `mysrc`, process it and save the result in `mydst`.

```
template parse args {one two} file mysrc file mydst
```

Will do the same as above, but the template code in the input file `mysrc` will have the list {one two} accessible via the `argv` variable.

```
template parse string "test" file mydst
```

Will process the string "test" and save the result in `mydst`.

Parameters:

args This optional argument should be followed by a list of arguments to pass to the template source file. It should be the very first argument, otherwise it is ignored. Each element of the list is available from the template source file in the `argv` array.

```
template link src dst
```

Link source file *src* to destination file *dst*. If relative, the source file *src* is interpreted as relative to the template directory and *dst* is interpreted as relative to the current output directory. Absolute file name can be given to override this behaviour.

```
template options { pattern body ...  }
```

Define the list of supported options for the template. Argument is a Tcl switch-like script that must define all supported options. It consists of pairs of *pattern body*. If an option matching the *pattern* is passed to the template, the *body* script is evaluated. A special body specified as "-" means that the body for the next pattern will be used for this pattern.

Examples:

```
template options {
-h - -help { puts "help option" }
}
```

This will make the template print the text "help option" whenever -h or –help option is passed to the template.

```
template arg
```

Return the next argument passed to the template, or raise an error is no argument remains.

```
template usage [string]
```

With a *string* argument, this procedure defines the template "usage" message. Unless the template redefines a `-h` option with `template options`, the default behaviour of the template is to print the content of the `template usage` string when -h or –help option is passed to the template.
`template usage`, when invoked without argument, returns the last usage message defined.

`template message [string]`

Print *string* so that it is visible to the end-user. The text is sent on the standard error channel unconditionnaly.

`template fatal [string]`

Print an error message and stop. In verbose mode, print the source location as reported by [info frame].

8.2 The engine command

`engine mode [ [+-]modespec ... ]`

Set miscellaneous engine operating mode. The command can be invoked without argument to retrieve the current settings for all supported modes. The command can also be invoked with one or more mode specification to set these modes (see modespec argument below). The list of supported modes is the following:

- `verbose`: turns on or off the verbosity of the engine.
- `overwrite`: when turned on, newly generated files will overwrite existing files without warning. When turned off, the engine will stop with an error if a newly generated file would overwrite an existing file. `overwrite` is by default off.
- `move-if-change`: when turned on, an existing file with the same content as a newly generated file will not be modified (preserving the last modification timestamp). When off, files are systematically updated. `move-if-change` is on by default.
- `debug`: when on, this mode preserves temporary files and tcl programs generated in the temporary directory. Useful only for debugging the template.

Example:

```
engine mode -overwrite +move-if-change
```

Parameters:

modespec	A mode specification string. Supported modes are `verbose`, `overwrite`, `move-if-change` and `debug`. If *mode* string is prefixed with a dash (-), it is turned off. If mode is prefixed with a plus (+) or nothing, it is turned on.

Returns:

When called without arguments, the command returs the current configuration of all engine modes.

```
engine chdir dir
```

Change the engine output directory. By default, files are generated in the current directory. This command can be used to generate output in any other directory.

Parameters:

dir	The new output directory, absolute or relative to the current working directory.

```
engine pwd
```

Return the current engine output directory.

8.3 The dotgen command

8.3.1 dotgen genom

Those commands implement access to genom program parameters or general information.

```
dotgen genom program
```

Returns the absolute path to the GenoM executable currently running.

```
dotgen genom cmdline
```

Returns a string containing the options passed to the GenoM program.

```
dotgen genom version
```

Returns the full version string of the GenoM program.

```
dotgen genom debug
```

Returns a boolean indicating whether genom was invoked in debugging mode or not.

```
dotgen genom stdout [on]
```

With optional boolean argument *on*, turns on or off the standard output channel of the template engine.
Without argument, the procedure returns the current activation status of the standard output channel.

Parameters:

[on]	Turn on/off standard output channel.

Returns:

When called without argument, returns a boolean indicating the current status (on/off) of the standard output channel.

8.3.2 dotgen template

Those commands return information about the template currently being parsed.

```
dotgen template name
```

Return the current template name.

```
dotgen template dir
```

Return a path to the current template directory (the directory holding the template.tcl file).

```
dotgen template sysdir
```

Return a path to the genom system directory.

```
dotgen template tmpdir
```

Return a path to the temporary directory where the template engine writes its temporary files.

8.3.3 dotgen input

Those commands return information about the current genom input file (.gen file).

```
dotgen input file
```

Return the full path to the current .gen file.

```
dotgen input base
```

Return the base name of the current .gen file, i.e. the file name with all directories stripped out.

```
dotgen input dir
```

Return the directory name of the current .gen file.

```
dotgen input notice
```

Return the copyright notice (as text) found in the .gen file. This notice can actually be any text and is the content of the special comment starting with the three caracters '/' '*' '/', near the beginning of the .gen file.

8.3.4 dotgen types

This command return information about the type definitions in the .gen file.

```
dotgen types [pattern]
```

This command returns the list of type objects that are defined in the current .gen file. This list may be filtered with the optional *pattern* argument. Each element of the returned list is a type command that can be used to access detailed information about that particular type object.

Parameters:

[pattern] Filter on the type names. The filter may contain a glob-like pattern (with * or ? wildcards). Only the types whose name match the pattern will be returned.

Returns:

A list of type objects.

8.3.5 dotgen components

This command return information about the components definitions in the .gen file.

```
dotgen components [pattern]
```

This command returns the list of components that are defined in the current .gen file (normally just one). This list may be filtered with the optional *pattern* argument. Each element of the returned list is a component command that can be used to access detailed information about each particular component object.

Parameters:

[pattern] Filter on the component name. The filter may contain a glob-like pattern (with * or ? wildcards). Only the components whose name match the pattern will be returned.

Returns:

A list of component objects.

8.4 The language command

```
language fileext lang [kind]
```

Return the cannonical file extension for the given language.

Parameters:

lang The language name. Must be `c` or `c++`.

kind Must be one of the strings `source` or `header`.

```
language comment lang text
```

Return a string that is a valid comment in the given language.

Parameters:

lang The language name. Must be `c`, `c++` or `shell`.

text The string to be commented.

```
language mapping lang [pattern]
```

Generate and return the mapping of the types matching the glob *pattern* (or all types if no pattern is given), for the given language. The returned string is a valid source code for the language.

Parameters:

lang The language name. Must be `c` or `c++`.

pattern A glob pattern to generate the mapping only for those type names that match the pattern.

```
language declarator lang type [var]
```

Return the abstract declarator for *type* or for a variable *var* of that type, in the given language.

Parameters:

lang	The language name. Must be `c` or `c++`.
type	A type object.
var	A string representing the name of a variable of type *type*.

```
language declarator& lang type [var]
```

Return the abstract declarator of a reference to the *type* or to a variable *var* of that type, in the given language.

Parameters:

lang	The language name. Must be `c` or `c++`.
type	A type object.
var	A string representing the name of a variable of type *type*.

```
language declarator* lang type [var]
```

Return the abstract declarator of a pointer to the *type* or to a variable *var* of that type, in the given language.

Parameters:

lang	The language name. Must be `c` or `c++`.
type	A type object.
var	A string representing the name of a variable of type *type*.

```
language reference lang type [var]
```

Return the expression representing a reference to a variable.

Parameters:

lang	The language name. Must be `c` or `c++`.
type	A type object.
var	A string representing the name of a variable of type *type*.

```
language dereference lang type [var]
```

Return the expression that dereferences a variable.

Parameters:

lang	The language name. Must be `c` or `c++`.
type	A type object.
var	A string representing the name of a variable of type *type*.

`language member` *lang type members*

Return the language construction to access a member of a *type*. If *members* is a list, it is interpreted as a hierachy. Each element of *members* is interpreted as follow: if it starts with a letter, *type* should be an aggregate type (like `struct`); if it starts with a numeric digit, *type* should be an array type (like `sequence`).
This procedure can typically be used with parameters objects, for which the 'member' subcommand returns a compatible *members* list.

Parameters:

lang	The language name. Must be `c` or `c++`.
type	A type object
members	A member name, or a list of hierachical members to access.

`language cname` *lang* {*string*|*object*}

Return the cannonical name of the *string* or the G[en]oM *object*, according to the language *lang*.
If a regular string is given, this procedure typically maps IDL :: scope separator into the native symbol in the given language. If a codel object is given, this procedure returns the symbol name of the codel in the given language.

Parameters:

lang	The language name. Must be `c` or `c++`.
string	The name to convert.
object	A G[en]oM object. Only class `codel` is supported at the moment.

`language signature lang codel` [*separator*]

Return the signature of a codel in the given language. If separator is given, it is a string that is inserted between the return type of the codel and the codel name (for instance, a \n in C so that the symbol name is guaranteed to be on the first column).

Parameters:

lang	The language name. Must be `c` or `c++`.
codel	A codel object.
separator	A string, inserted between the return type and the codel symbol name.

`language invoke` *lang codel params*

Return a string corresponding to the invocation of a codel in the given language.

Parameters:

lang	The language name. Must be `c` or `c++`.
codel	A codel object.
params	The list of parameters passed to the codel. Each element of this list must be a valid string in the *lang* language corresponding to each parameter value or reference to be passed to the codel.

```
language hfill text [filler] [column]
```

Return a string of length *column* (by default 80), starting with *text* and filled with the *filler* character (by default -).

Parameters:

text	The text to fill.
filler	The filler character (by default -).
column	The desired length of the returned string.

8.5 The object command

```
object foreach var object body
```

Evaluate *body* for each element of the G^{en}oM *object*, with *var* set to the current element. The result of the command is an empty string.
var is either a variable name, or a list of two variable names. In the latter case, the first variable is set to the current element and the second variable is set to a list of elements representing the hiearchy of objects leading to the current element.
If *object* is of class 'type' (a type object), the exploration of the type tree is done in depth-first order. If *body* returns with no error, the next element in explored. If *body* returns with 'continue', the exploration of the current branch is stopped and the next sibling element is explored. If *body* returns with 'break', the procedure returns immediately.
Note: when exploring 'type' objects, an endless recursion is possible if the type in question has a recursive definition (e.g. a structure contains a sequence of the same structure). This potentially endless recursion is allowed on purpose, but it is important that you handle this situation in the *body* script. A potentially endless recursion can be detected if your *body* script encounters either a `forward struct` or a `forward union`. It is up to the caller to determine what to do in this case, but this typically involves returning 'continue' at some point to skip further exploration of that branch.

Parameters:

var	A variable name that is set to the current element of object while iterating, or a list of two variable names respectively set to the current element and the hierarchy of elements leading to the current element.
object	object A genom object. Must be of class `type`.

body	A script evaluated for each element of object.

8.6 The buildenv command

8.6.1 buildenv autoconf

```
buildenv autoconf subdir dir
```

Add the subdirectory *dir* to the list of directories to be considered by the autoconf build environment. *dir* must be a subdirectory relative to the current output directory.
All the .m4 or .ac file found in *dir*/autoconf will be read by the main autoconf program (see buildenv autoconf create).

```
buildenv autoconf create
```

Create the main autoconf files, in the current output directory. Those files are common to all templates, but each template willing to use autoconf should call this procedure.

Bibliography

[1] Free Software Foundation. The C preprocessor. http://gcc.gnu.org/onlinedocs/cpp/.

[2] Object Management Group. CORBA specification, version 3.1. Part I: CORBA interfaces. http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF.

Index