GenoM Manual

For GenoM version 3

Revision 2.99.22, updated August 2013

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

Table of Contents

1

Introduction

2

Component model

3

GenoM overview

4

A minimal example

5

Input file format

This chapter describes the GenoM3 Input File Format (`dotgen`) semantics and gives the syntax for `dotgen` grammatical constructs. `dotgen` is the language used to formally describe a GenoM 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 5.1 [Preprocessing], page 9. The complete grammar is presented in Section 5.18 [Grammar reference], page 17. 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 EBNF (Extended Backus-Naur Format). The following table lists the symbols used in this format and their meaning.

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

5.1 Preprocessing

A `dotgen` specification consists of one or more files that are preprocessed. The preprocessing is controlled by directives introduced by lines having `#` as the first character other than white space. Preprocessor directives have their own syntax (namely, the C preprocessor syntax), independent of the `dotgen` language and not entirely described in this document. see *The C preprocessor*[1] for a comprehensive documentation.

The primary use of the preprocessing facility is to include definitions (especially type definitions) from other `dotgen` specifications. Directives may appear anywhere in the source file but are not seen nor interpreted by GenoM. For instance, text in files included with a `#include` directive is treated as if it appeared in the including file. However, some preprocessor `#pragma` directives are available to GenoM (see Section 5.17 [Pragmas], page 16).

The C preprocessor used by GenoM is `pcpp` from the `pcc` project (`http://pcc.ludd.ltu.se/`). It is invoked as a separate process from `libexec/genom-pcpp` by default. This can be changed by setting the environement variable `GENOM_CPP`, See Section 7.4 [Environment variables], page 39. However, note that if you change the default, you will loose some of the functionalities provided by `genom-pcpp`, like the `#pragma require` feature (see Section 5.17.1 [#pragma requires], page 17).

5.2 Elements of a GenoM3 specification

A dotgen specification consists of one or more statements. Statements are either GenoM statements, IDL statements. `cpp` directives (see Section 5.1 [Preprocessing], page 9) are handled at the lexical level and do not interfere with the specification grammar.

[1] `http://gcc.gnu.org/onlinedocs/cpp/`

```
(1) specification          ::= { statement }
(2) statement              ::= component
                             | interface
                             | idl-statement

(4) idl-statement          ::= module
                             | const-dcl
                             | type-dcl
```

Definitions are named by the mean of identifiers, see Section 5.16 [Reserved keywords], page 15.

A GenoM statement defines components (see Section 5.3 [Component declaration], page 10) or interfaces (see Section 5.4 [Interface declaration], page 11).

An IDL statement defines types (see Section 5.14 [Type declaration], page 15), constants (see Section 5.13 [Constant declaration], page 14) or IDL modules containing types and constants (see Section 5.12 [Module declaration], page 14). The syntax follows closely the subset the OMG IDL specification corresponding to type and constants definitions (see Chapter 7 of *CORBA specification, Object Management Group, version 3.1. Part I: CORBA interfaces*). Note that this subset of the dogten grammar is not in any manner tied to OMG IDL and may diverge from future OMG specifications.

5.3 Component declaration

```
(5) component              ::= "component" component-name component-body ";"
(6) component-name         ::= identifier
(7) component-body         ::= [ "{" exports "}" ]

(8) exports                ::= { export }
(9) export                 ::= idl-statement
                             | property
                             | ids
                             | task
                             | port
                             | attribute
                             | service

(10) component-property    ::= ( "doc" string-literals | "version"
                               string-literals | "lang" string-literals |
                               "email" string-literals | "requires"
                               string-list | "codels-require" string-list
                               | "clock-rate" const-expr time-unit |
                                 "provides" interface-list | "uses"
                                 interface-list ) ";"
```

A component declaration describes a instance of the GenoM component model. It is defined by a unique name (an identifier) that also defines an IDL scope for any embedded types.

Components export objects from the GenoM component model, namely: IDS (see Section 5.5 [IDS declaration], page 11), tasks (see Section 5.6 [Task declaration], page 11), ports (see Section 5.7 [Port declaration], page 12), attributes (see Section 5.8 [Attribute declaration], page 12) or services (see Section 5.9 [Service declaration], page 13).

Components may also define new types *via* IDL statements. These types will be defined within the component scope.

A number of properties can be attached to a component:

`doc` A string that describes the functionality of the component.

`version` The component version number, as a string

`lang` The programming language of the codels interface.

`email` A string containing the e-mail address of the author of the component.

`requires` A list of dependencies of the component (see Section 5.17.1 [#pragma requires], page 17). Each string should contain a package name in `pkg-config` format.

`codels-requires`
A list of dependencies of the codels. Each string should contain a package name in `pkg-config` format.

`clock-rate`
The period of the internal component clock. It is usually not necessary to define it explicitly. If the component defines periodic task, the component clock period will be automatically computed as the greatest common divisor of the period of all periodic tasks.

`provides` A list of interfaces (see Section 5.4 [Interface declaration], page 11) that the component implements. All objects from the interface are imported as-is into the component description. Ports and services may be further refined once imported, typically by defining codels (see Section 5.11 [Codel declaration], page 14) that implement the services.

`uses` A list of interfaces (see Section 5.4 [Interface declaration], page 11) that the component uses. Ports are imported in the opposite direction (e.g. a `port out` is imported as a `port in`. Services are imported as `remote` objects that can be accessed *via* codel parameters (see Section 5.11 [Codel declaration], page 14). Other objects are imported as-is.

5.4 Interface declaration

```
(13) interface             ::= "interface" interface-scope component-body ";"
(14) interface-scope       ::= identifier

(16) interface-property    ::= "extends" interface-list ";"
```

An interface declaration is basically the same as a component declaration (see Section 5.3 [Component declaration], page 10) but is meant to be shared between several components. Although any object can be defined in an interface, it will typically only declare service prototypes and ports that are to be `provided` or `used` by components.

In addition to regular component properties, an interface can also define the following properties:

`extends` A list of interfaces that are imported as-is into the current one. All objects from the extended interfaces appear as if they had been defined in the extending interface.

5.5 IDS declaration

```
(18) ids                   ::= ids-name "{" member-list "}" ";"
(19) ids-name              ::= "ids"
```

5.6 Task declaration

```
(20) task                  ::= "task" identifier opt-properties ";"
(122) opt-properties       ::= [ "{" properties "}" ]
(123) properties           ::= { property }

(21) task-property         ::= ( "period" const-expr time-unit | "delay"
                               const-expr time-unit | "priority"
```

```
                                positive-int-const | "scheduling" "real-time" |
                                "stack" positive-int-const size-unit ) ";"
(40) codel-property         ::= opt-async "codel" ( codel ";" | fsm-codel ";" )
```

Tasks define an execution context suitable for running *activities* (see Section 5.3 [Component declaration], page 10). A task may define a state machine and associated codels (see Section 5.11 [Codel declaration], page 14). The state machine starts in the `start` state when the task is created during component initialization.

Tasks are named can also define the following properties:

`period` The granularity of the codel scheduler. Periodic task will sequence the codels they manage at that frequency.

`delay` The delay from the beginning of each period after which codels are run. This can be used to delay two tasks running at the same period in the same component.

`priority` Can be used to prioritize different tasks whithin the same component.

`scheduling real-time`
This indicates that the task requires real-time scheduling. This may not be supported by all templates.

`stack` Defines the required stack size for this task. The stack size should be big enough to run all codels that the task manages.

5.7 Port declaration

```
(22) port                   ::= "port" opt-multiple port-dir type-spec
                                identifier ";"
(24) opt-multiple           ::= [ "multiple" ]
(23) port-dir               ::= "in"
                              | "out"
```

Ports implement the data flow between components as a publish/subscribe model. Ports have a name and a type and can be either `out` (for publishing data) or `in` (for subscribing to a sibling `out` port).

The optional `multiple` qualifier defines a dynamic list of ports of the given type, indexed by strings. In this case, ports are created or destroyed dynamically be the codels.

5.8 Attribute declaration

```
(25) attribute              ::= "attribute" identifier "(" attribute-parameters
                                ")" opt-properties ";"
(29) attribute-parameters   ::= [ { attribute-parameter "," }
                                attribute-parameter ]
(30) attribute-parameter    ::= parameter-dir parameter-variable
                                opt-initializer
(122) opt-properties        ::= [ "{" properties "}" ]
(123) properties            ::= { property }

(28) service-property       ::= ( "task" identifier | "interrupts"
                                identifier-list | "before" identifier-list |
                                "after" identifier-list | "validate" codel |
                                "local" local-variables ) ";"
```

5.9 Service declaration

```
 (26) service              ::= service-kind identifier "(" service-parameters
                               ")" opt-properties ";"
 (27) service-kind         ::= "function"
                             | "activity"
 (31) service-parameters   ::= [ { service-parameter "," } service-parameter ]
 (32) service-parameter    ::= parameter-dir type-spec declarator
                               opt-initializer
(122) opt-properties       ::= [ "{" properties "}" ]
(123) properties           ::= { property }
(124) property             ::= component-property
                             | interface-property
                             | task-property
                             | service-property
                             | codel-property
                             | throw-property

 (28) service-property     ::= ( "task" identifier | "interrupts"
                               identifier-list | "before" identifier-list |
                               "after" identifier-list | "validate" codel |
                               "local" local-variables ) ";"
 (40) codel-property       ::= opt-async "codel" ( codel ";" | fsm-codel ";" )
 (36) opt-async            ::= [ "async" ]
```

5.10 Service parameters

```
 (30) attribute-parameter  ::= parameter-dir parameter-variable
                               opt-initializer
 (32) service-parameter    ::= parameter-dir type-spec declarator
                               opt-initializer

 (42) parameter-dir        ::= "in"
                             | "out"
                             | "inout"
 (43) parameter-variable   ::= identifier
                             | parameter-variable "." identifier
                             | parameter-variable "[" positive-int-const "]"
 (44) opt-initializer      ::= [ "=" initializer ]
 (45) initializers         ::= [ { initializer "," } initializer ]
 (46) initializer          ::= initializer-value
                             | ":" string-literals
                             | initializer-value ":" string-literals
 (47) initializer-value    ::= const-expr
                             | "{" initializers "}"
                             | "[" positive-int-const "]" "=" const-expr
                             | "[" positive-int-const "]" "=" "{" initializers
                               "}"
                             | "[" positive-int-const "]" "="
                             | "." identifier "=" const-expr
```

```
                              | "." identifier "=" "{" initializers "}"
                              | "." identifier "="
```

5.11 Codel declaration

```
(34) codel                ::= identifier "(" codel-parameters ")"

(35) fsm-codel            ::= "<" event-list ">" identifier "("
                              codel-parameters ")" "yields" event-list

(38) codel-parameters     ::= [ { codel-parameter "," } codel-parameter ]

(39) codel-parameter      ::= opt-parameter-src parameter-dir (
                              parameter-variable
                              | parameter-variable "::" identifier | "::"
                                identifier )

(36) opt-async            ::= [ "async" ]

(41) opt-parameter-src    ::= [ "ids" | "local" | "port" | "remote" ]

(42) parameter-dir        ::= "in"
                              | "out"
                              | "inout"

(43) parameter-variable   ::= identifier
                              | parameter-variable "." identifier
                              | parameter-variable "[" positive-int-const "]"

(37) event-list           ::= { scoped-name "," } scoped-name
```

5.12 Module declaration

A module definition satisfies the following syntax:

```
(48) module               ::= "module" module-name "{" module-body "}" ";"

(49) module-name          ::= identifier

(50) module-body          ::= [ idl-statements ]

 (3) idl-statements       ::= { idl-statement } idl-statement
```

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.

5.13 Constant declaration

```
(55) const-dcl            ::= "const" const-type identifier "=" const-expr
                              ";"

(56) const-type           ::= integer-type
                              | char-type
                              | boolean-type
                              | floating-pt-type
                              | octet-type
                              | string-type
                              | named-type
```

5.14 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:

```
(57) type-dcl              ::= constructed-type ";"
                             | "typedef" alias-list ";"
                             | "native" identifier ";"
                             | EXCEPTION exception-list ";"
                             | forward-dcl
(58) constructed-type      ::= struct-type
                             | union-type
                             | enum-type
(59) alias-list            ::= ( type-spec | alias-list "," ) declarator
```

5.15 Type specification

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

```
(70) type-spec             ::= simple-type-spec
                             | constructed-type-spec
(71) simple-type-spec      ::= base-type-spec
                             | template-type-spec
                             | named-type
(74) base-type-spec        ::= boolean-type
                             | integer-type
                             | floating-pt-type
                             | char-type
                             | octet-type
                             | any-type
(75) template-type-spec    ::= sequence-type
                             | string-type
                             | fixed-type
(72) constructed-type-spec ::= constructed-type
(73) named-type            ::= scoped-name
(54) scoped-name           ::= [ [ scoped-name ] "::" ] identifier
(66) declarator            ::= simple-declarator
                             | array-declarator
(67) simple-declarator     ::= identifier
```

5.16 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.

```
(105) identifier           ::= "[A-Za-z-][A-Za-z0-9-]*"
                             | "s"
                             | "ms"
                             | "us"
                             | "k"
                             | "m"
```

```
                                | "real-time"
                                | "interface"
                                | "component"
                                | "ids"
                                | "attribute"
                                | "function"
                                | "activity"
                                | "version"
                                | "lang"
                                | "email"
                                | "requires"
                                | "codels-require"
                                | "clock-rate"
                                | "task"
                                | "task"
                                | "period"
                                | "delay"
                                | "priority"
                                | "scheduling"
                                | "stack"
                                | "codel"
                                | "validate"
                                | "yields"
                                | "throws"
                                | "doc"
                                | "interrupts"
                                | "before"
                                | "after"
                                | "handle"
                                | "port"
                                | "in"
                                | "out"
                                | "inout"
                                | "local"
                                | "async"
                                | "remote"
                                | "extends"
                                | "provides"
                                | "uses"
                                | "multiple"
                                | "native"
                                | EXCEPTION

(106) identifier-list       ::= { identifier "," } identifier
```

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

5.17 Pragmas

Pragmas are a method for providing additional information to GenoM, beyond what is conveyed in the language itself. They are introduced by the `#pragma` directive, followed by arguments. GenoM understands the following pragmas:

5.17.1 `#pragma requires` directives

`#pragma requires` is recognized by *both* `genom-pcpp` preprocessor and GenoM. It indicates an external dependency on a software package that is required to parse the current specification. `#pragma requires` assumes that the package is using the `pkg-config` utility (see `http://www.freedesktop.org/wiki/Software/pkg-config`) and a `.pc` is available. This has the same effect as placing `requires` directives in all components (see Section 5.3 [Component declaration], page 10) but saves the need pass `-I` and `-D` directives to `genom` (see Section 7.2 [General options], page 37) as they are automatically computed.

The pragma syntax is as follow:

```
#pragma requires "package [ >= version ]"
```

`#pragma requires` accepts a string argument in the form `package [>= version]`. `genom-pcpp` interprets it by running `pkg-config --cflags` on the string argument. It then adds the resulting `-I` and `-D` flags as if they had been passed on the command line. Note that the flags are added *at the current processing location*, so they do not influence already preprocessed input. The `pkg-config` utility is found in `PATH`, or via the `PKG_CONFIG` environment variable if defined (see Section 7.4 [Environment variables], page 39).

The pragma argument is also added to the `require` property of *all* the components defined in a specification.

5.17.2 `#pragma provides` directives

`#pragma provides` achieves the same effect as if *all* components of a specification defined the same `provides` property (see Section 5.3 [Component declaration], page 10). This directive is mostly useful for templates implementation, so that they can provide a common interface to all user defined components.

The pragma syntax is as follow:

```
#pragma provides interface
```

5.17.3 `#pragma masquerade` directives

This directive applies to an IDL type definition in a component interface. It is meant for aliasing the IDL type description to a native object that cannot be described in IDL. The exact nature of the native object depends on the template used for code generation, so it is only described as a raw string here and not interpreted by GenoM.

The pragma syntax is as follow:

```
#pragma masquerade template type data...
```

The *template* argument is a free form string that indicates to which template the directives applies. Templates can lookup this name and take the appropriate actions based on this information. *type* is the name of the IDL type thas is to be masqueraded. *data* describes how the masquerading will be done, and is template specific. Refer to the documentation of the template you are using for a precise description of the syntax and semantics of *data*.

5.18 Grammar reference

```
(1) specification       ::= { statement }
(2) statement           ::= component
                          | interface
                          | idl-statement
(3) idl-statements      ::= { idl-statement } idl-statement
(4) idl-statement       ::= module
                          | const-dcl
                          | type-dcl
(5) component           ::= "component" component-name component-body ";"
```

```
 (6) component-name        ::= identifier
 (7) component-body        ::= [ "{" exports "}" ]
 (8) exports               ::= { export }
 (9) export                ::= idl-statement
                             | property
                             | ids
                             | task
                             | port
                             | attribute
                             | service
(10) component-property    ::= ( "doc" string-literals | "version"
                               string-literals | "lang" string-literals |
                               "email" string-literals | "requires"
                               string-list | "codels-require" string-list
                               | "clock-rate" const-expr time-unit |
                                 "provides" interface-list | "uses"
                                 interface-list ) ";"
(11) throw-property        ::= "throws" throw-list ";"
(12) throw-list            ::= { named-type "," } named-type
(13) interface             ::= "interface" interface-scope component-body ";"
(14) interface-scope       ::= identifier
(15) interface-name        ::= identifier
(16) interface-property    ::= "extends" interface-list ";"
(17) interface-list        ::= { interface-name "," } interface-name
(18) ids                   ::= ids-name "{" member-list "}" ";"
(19) ids-name              ::= "ids"
(20) task                  ::= "task" identifier opt-properties ";"
(21) task-property         ::= ( "period" const-expr time-unit | "delay"
                               const-expr time-unit | "priority"
                               positive-int-const | "scheduling" "real-time" |
                               "stack" positive-int-const size-unit ) ";"
(22) port                  ::= "port" opt-multiple port-dir type-spec
                               identifier ";"
(23) port-dir              ::= "in"
                             | "out"
(24) opt-multiple          ::= [ "multiple" ]
(25) attribute             ::= "attribute" identifier "(" attribute-parameters
                               ")" opt-properties ";"
(26) service               ::= service-kind identifier "(" service-parameters
                               ")" opt-properties ";"
(27) service-kind          ::= "function"
                             | "activity"
(28) service-property      ::= ( "task" identifier | "interrupts"
                               identifier-list | "before" identifier-list |
                               "after" identifier-list | "validate" codel |
                               "local" local-variables ) ";"
```

```
(29) attribute-parameters  ::= [ { attribute-parameter "," }
                               attribute-parameter ]
(30) attribute-parameter   ::= parameter-dir parameter-variable
                               opt-initializer
(31) service-parameters    ::= [ { service-parameter "," } service-parameter ]
(32) service-parameter     ::= parameter-dir type-spec declarator
                               opt-initializer
(33) local-variables       ::= ( type-spec | local-variables "," ) declarator
(34) codel                 ::= identifier "(" codel-parameters ")"
(35) fsm-codel             ::= "<" event-list ">" identifier "("
                               codel-parameters ")" "yields" event-list
(36) opt-async             ::= [ "async" ]
(37) event-list            ::= { scoped-name "," } scoped-name
(38) codel-parameters      ::= [ { codel-parameter "," } codel-parameter ]
(39) codel-parameter       ::= opt-parameter-src parameter-dir (
                               parameter-variable
                               | parameter-variable "::" identifier | "::"
                                 identifier )
(40) codel-property        ::= opt-async "codel" ( codel ";" | fsm-codel ";" )
(41) opt-parameter-src     ::= [ "ids" | "local" | "port" | "remote" ]
(42) parameter-dir         ::= "in"
                             | "out"
                             | "inout"
(43) parameter-variable    ::= identifier
                             | parameter-variable "." identifier
                             | parameter-variable "[" positive-int-const "]"
(44) opt-initializer       ::= [ "=" initializer ]
(45) initializers          ::= [ { initializer "," } initializer ]
(46) initializer           ::= initializer-value
                             | ":" string-literals
                             | initializer-value ":" string-literals
(47) initializer-value     ::= const-expr
                             | "{" initializers "}"
                             | "[" positive-int-const "]" "=" const-expr
                             | "[" positive-int-const "]" "=" "{" initializers
                               "}"
                             | "[" positive-int-const "]" "="
                             | "." identifier "=" const-expr
                             | "." identifier "=" "{" initializers "}"
                             | "." identifier "="
(48) module                ::= "module" module-name "{" module-body "}" ";"
(49) module-name           ::= identifier
(50) module-body           ::= [ idl-statements ]
(51) scope-push-struct     ::= identifier
(52) scope-push-union      ::= identifier
(53) exception-name        ::= identifier
```

```
(54) scoped-name           ::= [ [ scoped-name ] "::" ] identifier
(55) const-dcl             ::= "const" const-type identifier "=" const-expr
                               ";"
(56) const-type            ::= integer-type
                             | char-type
                             | boolean-type
                             | floating-pt-type
                             | octet-type
                             | string-type
                             | named-type
(57) type-dcl              ::= constructed-type ";"
                             | "typedef" alias-list ";"
                             | "native" identifier ";"
                             | EXCEPTION exception-list ";"
                             | forward-dcl
(58) constructed-type      ::= struct-type
                             | union-type
                             | enum-type
(59) alias-list            ::= ( type-spec | alias-list "," ) declarator
(60) struct-type           ::= "struct" scope-push-struct "{" member-list "}"
(61) union-type            ::= "union" scope-push-union "switch" "("
                               switch-type-spec ")" "{" switch-body "}"
(62) exception-list        ::= { exception-dcl "," } exception-dcl
(63) exception-dcl         ::= exception-name opt-member-list
(64) enum-type             ::= "enum" identifier "{" enumerator-list "}"
(65) forward-dcl           ::= ( "struct" | "union" ) identifier ";"
(66) declarator            ::= simple-declarator
                             | array-declarator
(67) simple-declarator     ::= identifier
(68) array-declarator      ::= ( simple-declarator | array-declarator )
                               fixed-array-size
(69) fixed-array-size      ::= "[" positive-int-const "]"
(70) type-spec             ::= simple-type-spec
                             | constructed-type-spec
(71) simple-type-spec      ::= base-type-spec
                             | template-type-spec
                             | named-type
(72) constructed-type-spec ::= constructed-type
(73) named-type            ::= scoped-name
(74) base-type-spec        ::= boolean-type
                             | integer-type
                             | floating-pt-type
                             | char-type
                             | octet-type
                             | any-type
(75) template-type-spec    ::= sequence-type
                             | string-type
                             | fixed-type
```

```
 (76) integer-type          ::= signed-int
                              | unsigned-int
 (77) floating-pt-type      ::= float-type
                              | double-type
 (78) signed-int            ::= signed-longlong-int
                              | signed-long-int
                              | signed-short-int
 (79) unsigned-int          ::= unsigned-longlong-int
                              | unsigned-long-int
                              | unsigned-short-int
 (80) unsigned-short-int    ::= "unsigned" "short"
 (81) unsigned-long-int     ::= "unsigned" "long"
 (82) unsigned-longlong-int ::= "unsigned" "long" "long"
 (83) signed-short-int      ::= "short"
 (84) signed-long-int       ::= "long"
 (85) signed-longlong-int   ::= "long" "long"
 (86) float-type            ::= "float"
 (87) double-type           ::= "double"
 (88) char-type             ::= "char"
 (89) boolean-type          ::= "boolean"
 (90) octet-type            ::= "octet"
 (91) any-type              ::= "any"
 (92) string-type           ::= "string" [ "<" positive-int-const ">" ]
 (93) sequence-type         ::= "sequence" "<" simple-type-spec ( ","
                                positive-int-const ">" | ">" )
 (94) fixed-type            ::= "fixed" [ "<" positive-int-const ","
                                positive-int-const ">" ]
 (95) switch-type-spec      ::= integer-type
                              | char-type
                              | boolean-type
                              | enum-type
                              | named-type
 (96) switch-body           ::= { case } case
 (97) opt-member-list       ::= [ "{" ( "}" | member-list "}" ) ]
 (98) member-list           ::= { member ";" } member ";"
 (99) member                ::= ( type-spec | member "," ) declarator
(100) case                  ::= case-label-list type-spec declarator ";"
(101) case-label-list       ::= { case-label } case-label
(102) case-label            ::= ( "case" const-expr | "default" ) ":"
(103) enumerator-list       ::= { enumerator "," } enumerator
(104) enumerator            ::= identifier
(105) identifier            ::= "[A-Za-z-][A-Za-z0-9-]*"
                              | "s"
                              | "ms"
                              | "us"
```

```
                                | "k"
                                | "m"
                                | "real-time"
                                | "interface"
                                | "component"
                                | "ids"
                                | "attribute"
                                | "function"
                                | "activity"
                                | "version"
                                | "lang"
                                | "email"
                                | "requires"
                                | "codels-require"
                                | "clock-rate"
                                | "task"
                                | "task"
                                | "period"
                                | "delay"
                                | "priority"
                                | "scheduling"
                                | "stack"
                                | "codel"
                                | "validate"
                                | "yields"
                                | "throws"
                                | "doc"
                                | "interrupts"
                                | "before"
                                | "after"
                                | "handle"
                                | "port"
                                | "in"
                                | "out"
                                | "inout"
                                | "local"
                                | "async"
                                | "remote"
                                | "extends"
                                | "provides"
                                | "uses"
                                | "multiple"
                                | "native"
                                | EXCEPTION
(106) identifier-list       ::= { identifier "," } identifier
(107) const-expr            ::= or-expr
(108) positive-int-const    ::= const-expr
(109) or-expr               ::= { xor-expr "|" } xor-expr
(110) xor-expr              ::= { and-expr "^" } and-expr
(111) and-expr              ::= { shift-expr "&" } shift-expr
```

```
(112) shift-expr            ::= { add-expr ( ">>" | "<<" ) } add-expr
(113) add-expr              ::= { mult-expr ( "+" | "-" ) } mult-expr
(114) mult-expr             ::= { unary-expr ( "*" | "/" | "%" ) } unary-expr
(115) unary-expr            ::= [ "-" | "+" | "~" ] primary-expr
(116) primary-expr          ::= literal
                              | "(" const-expr ")"
                              | named-type
(117) literal               ::= "TRUE"
                              | "FALSE"
                              | integer-literal
                              | "<float-literal>"
                              | "<fixed-literal>"
                              | "<char-literal>"
                              | string-literals
(118) string-literals       ::= { string-literal } string-literal
(119) string-list           ::= { string-literals "," } string-literals
(120) time-unit             ::= [ "s" | "ms" | "us" ]
(121) size-unit             ::= [ "k" | "m" ]
(122) opt-properties        ::= [ "{" properties "}" ]
(123) properties            ::= { property }
(124) property              ::= component-property
                              | interface-property
                              | task-property
                              | service-property
                              | codel-property
                              | throw-property
```

6

GenoM IDL mappings

GenoM IDL is independent of the programming language used to implement the services and internals of a component. In order to use the GenoM 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 GenoM IDL constructs to the supported programming languages.

The mapping between GenoM 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.

GenoM currently implements mappings for the `C` and `C++` languages. For the `C` language, See Section 6.1 [C mappings], page 25. For the `C++` language, see See Section 6.2 [C++ mappings], page 30.

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 GenoM 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 mapped to a `C` constant. For instance, the following IDL:

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

would map into

```
const uint32_t longint = 1;
const char *str = "string example";
```

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

6.1.3 Mapping for basic data types

The basic data types have the mappings shown in the table below. 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.

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	int8_t
octet	uint8_t
any	type any not implemented yet

Table: Basic data types mappings in C

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 in an anonymous `enum` with an appropriate unsigned integer value conforming to the ordering constraints.

For instance, the following IDL:

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

would map, according to the scoped names rules, into

```
typedef uint32_t m_e;
enum {
  m_value1 = 0
  m_value2 = 1
};
```

6.1.5 Mapping for strings

GenoM 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

GenoM 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

GenoM 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

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

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

6.1.9 Mapping for sequence types

GenoM 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;
```

A helper function for reserving space in unbounded sequences is available. It is defined as follow:

```
int genom_sequence_reserve(sequence_type *s, uint32_t length);
```

where `sequence_type` is the actual type name of the sequence.

Given a pointer `s` on a valid, unbouded sequence, `genom_sequence_reserve` will allocate storage for up to `length` elements. If needed, the previous storage is first released using the function pointed to by `_release`. If the sequence already had enough space available, `genom_sequence_reserve` does nothing. New storage is allocated using `malloc()`, and the `_release` member is updated to point on the `free()` function. The `_maximum` member is updated to reflect the new maximum number of elements that the sequence may hold. Finally, the function returns 0 on success, or -1 if no memory could be allocated (in this case, the global variable `errno` is updated with the value `ENOMEM`).

Beware that `genom_sequence_reserve` is able to shrink a sequence if the `length` parameter happens to be smaller than the current length of a sequence. In this case, any elements previously stored beyond the new effective length will be definitely lost. The `_length` member will be updated accordingly.

The use of this function to reserve space in a sequence is purely optional. Any storage allocation strategy can be used, provided it maintains the consistency between the `_maximum`, `_release` and `_buffer` elements.

6.1.10 Mapping for port types

Simple ports map onto an object-like C `struct` with a `data()` and `read()` or `write()` function members. The `data()` function takes no parameter and returns a pointer on the current port data. Input ports may refresh their data by invoking the `read()` method, while output ports may publish new data by invoking the `write()` method. Both `read()` and `write()` return `genom_ok` on success, or a `genom_event` exception representing an error code.

Ports defined with the `multiple` flag map onto a similar `struct`, with the difference that `data()`, `read()` and `write()` methods take an additional string (`const char *`) parameter representing the port element name. Multiple output ports have two additional `open()` and `close()` members (also accepting a single string parameter) that dynamically create or destroy ports.

For instance, the following IDL:

```
port in double in_port;
port multiple in double multi_in_port;
port out double out_port;
port multiple out double multi_out_port;
```

would map into

```
typedef struct {
    double * (*data)();
    genom_event (*read)(void);
} in_port;

typedef struct {
    double * (*data)(const char *id);
```

```
   genom_event (*read)(const char *id);
} multi_in_port;

typedef struct {
   double * (*data)();
   genom_event (*write)(void);
} out_port;

typedef struct {
   double * (*data)(const char *id);
   genom_event (*write)(const char *id);
   genom_event (*open)(const char *id);
   genom_event (*close)(const char *id);
} multi_out_port;
```

6.1.11 Mapping for remote services

Remote objects map onto an object-like C `struct` providing a `call()` method. `call()` takes the same parameters as the corresponing service definition and returns `genom_ok` on success, or a `genom_event` exception representing an error code.

For instance, the following IDL:

```
interface i {
  function f(in long i, out double o);
};

component c {
  uses i;
};
```

would map into

```
typedef struct c_f {
  genom_event (*call)(uint32_t i, double *o);
} c_f;
```

The remote service is invoked in a synchronous manner.

6.1.12 Mapping for native types

GenoM IDL native types map to a C `struct`. The mapping provides only a forward declaration, and the user has to provide the actual definition.

For instance, the following IDL:

```
native opaque;
```

would map into

```
typedef struct opaque opaque;
```

The definition of the structure body is free, and will typically use native C types that cannot be described in IDL. When used as a parameter of a function, a native type will be passed around as a pointer on the structure data. Memory management associated with that pointer must be handled by the user.

6.1.13 Mapping for exceptions

Each defined exception type is defined as a `struct` tag and a `typedef` with the C global name for the exception suffixed by `_detail`. An identifier for the exception is also defined, as is a type-specific function for raising the exception. For example:

```
exception foo {
  long dummy;
};
```

yields the following C declarations:

```
genom_event foo_id = <unique identifier for exception>;

typedef struct foo_detail {
  uint32_t dummy;
} foo_detail;

genom_event foo(const foo_detail *detail);
```

The identifier for the exception uniquely identifies this exception type, so that any data of type `genom_event` can be compared to an exception id with the `==` operator.

The function throwing the exception returns a `genom_event` that should be used as the return value of a codel. It makes a copy of the exception details.

Since exceptions are allowed to have no members, but C structs must have at least one member, exceptions with no members map to the C `void` type and the type-specific throw function takes no argument.

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";
}
```

6.2.2 Mapping for constants

GenoM 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 the table below. 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.

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	int8_t
octet	uint8_t
any	type any not implemented yet

Table: Basic data types mappings in C++

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 {
   value1,
   value2,
   _unused = 0xffffffff,
};
```

6.2.5 Mapping for strings

GenoM IDL bounded strings are mapped to nul terminated character arrays (i.e., C strings). 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 char bounded[16];
```

6.2.6 Mapping for arrays

GenoM 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

GenoM 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 sequences maps onto the `std::vector` template class provided by the C++ standard. The bounded sequence maps onto a C++ `genom::bounded_sequence` template class. The definition of `genom::bounded_sequence` is very similar to `std::array` but provides a variable number of elements.

For instance, the following IDL:

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

would map into

```
typedef std::vector<int32_t> unbounded;
typedef genom::bounded_sequence<int32_t, 16> bounded;
```

The interface of `genom::bounded_sequence` is the following:

```
template <typename T, size_t N>
struct bounded_sequence {
  // types:
  typedef T                                     value_type;
  typedef value_type&                           reference;
  typedef const value_type&                     const_reference;
  typedef value_type*                           iterator;
  typedef const value_type*                     const_iterator;
  typedef value_type*                           pointer;
  typedef const value_type*                     const_pointer;
  typedef size_t                                size_type;
  typedef ptrdiff_t                             difference_type;
  typedef std::reverse_iterator<iterator>       reverse_iterator;
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

  value_type e[N];
  size_type n;

  // No explicit construct/copy/destroy for aggregate type

  void fill(const value_type &u);

  // iterators:
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;

  reverse_iterator rbegin();
  reverse_iterator rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;

  const_iterator cbegin() const;
  const_iterator cend() const;
  const_reverse_iterator crbegin() const;
  const_reverse_iterator crend() const;

  // capacity:
  size_type size() const;
  void resize(size_type l, value_type u = value_type());
  size_type max_size() const;
  bool empty() const;

  // element access:
  reference operator[](size_type i);
  const_reference operator[](size_type i) const;
  reference at(size_type i);
  const_reference at(size_type i) const;

  reference front();
  const_reference front() const;
```

```
    reference back();
    const_reference back() const;
    value_type *data();
    const value_type *data() const;

    // modifiers
    void swap(bounded_sequence &a);
    void clear();
  };
```

6.2.10 Mapping for port types

Simple ports map onto a pure virtual `struct` providing a `data()` and `read()` or `write()` methods. The `data()` method takes no parameter and returns a constant reference on the current port data. Input ports may refresh their data by invoking the `read()` method, while output ports may publish new data by invoking the `write()` method. Both `read()` and `write()` return no value (`void`).

Ports defined with the `multiple` flag map onto a similar pure virtual `struct`, with the difference that `data()`, `read()` and `write()` methods take an additional string (`const char *`) parameter representing the port element name. Multiple output ports have two additional `open()` and `close()` members (also accepting a single string parameter) that dynamically create or destroy ports.

All these method may throw a `genom::exception` representing an error code.

For instance, the following IDL:

```
port in double in_port;
port multiple in double multi_in_port;
port out double out_port;
port multiple out double multi_out_port;
```

would map into

```
struct in_port {
   virtual const double &data(void) const = 0;
   virtual void read(void) = 0;
};

struct multi_in_port {
   virtual const double &data(const char *id) const = 0;
   virtual void read(const char *id) = 0;
};

struct out_port {
   virtual double &data(void) const = 0;
   virtual void write(void) = 0;
};

struct multi_out_port {
   virtual double &data(const char *id) const = 0;
   virtual void write(const char *id) = 0;
   virtual void open(const char *id) = 0;
   virtual void close(const char *id) = 0;
};
```

6.2.11 Mapping for remote services

Remote objects map onto a pure virtual `struct` providing a `call()` method. `call()` takes the same parameters as the corresponing service definition and return no data (`void`). It may throw a `genom::exception` representing an error code.

For instance, the following IDL:

```
interface i {
  function f(in long i, out double o);
};

component c {
  uses i;
};
```

would map into

```
namespace c {
  struct f {
    virtual void call(uint32_t i, double &o) = 0;
  };
}
```

The remote service is invoked in a synchronous manner.

6.2.12 Mapping for native types

GenoM IDL native types map to a C++ `struct`. The mapping provides only a forward declaration, and the user has to provide the actual definition.

For instance, the following IDL:

```
native opaque;
```

would map into

```
struct opaque;
```

The definition of the structure body is free, and will typically use native C++ types that cannot be described in IDL. When used as a parameter of a function, a native type will be passed around as a pointer on the structure data. Memory management associated with that pointer must be handled by the user.

6.2.13 Mapping for exceptions

Each defined exception type is defined as a C++ `struct` that derives from the generic `genom::exception` type and implements a `what()` method returning a unique identifier for the exception. Exceptions with members define an additional `struct detail` type inside the scope of the exception as well a a `detail` member of that type. A global identifier for the exception is also defined (it is identical to the return value of the `what` method).

For example:

```
exception foo {
  long dummy;
};
```

yields the following C++ declarations:

```
genom_event foo_id = <unique identifier for exception>;

struct foo : public genom::exception {
  struct detail {
    uint32_t dummy;
```

```
    } detail;

    const char *what();
  } foo_detail;
```

Exceptions must be thrown with the C++ `throw` operator.

The identifier for the exception uniquely identifies this exception type, so that any data of type `genom_event` can be compared to an exception id with the `==` operator.

7

Running GenoM

GenoM is invoked by using one of the three following command lines:

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

The following sections give an overview of the general behaviour (see Section 7.1 [Description], page 37) and detail the options affecting the GenoM program itself (see Section 7.2 [General options], page 37) as well as options that can be passed to templates (see Section 7.3 [Template options], page 39). A list of recognized environment variables is also given (see Section 7.4 [Environment variables], page 39).

7.1 Description

`genom3` generates the source code of the software components described in the formal description *file.gen* input file.

The input *file.gen* 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 `genom3` and transformed into an abstract syntax tree. `libexec/genom-pcpp` is the default program, but this can be changed with the `CPP` environment variable. `genom3` 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. They are interpreted by the generator engine, and contain either code written with a scripting language, 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 `genom3` program accepts *general options* that affect the general program behaviour. `genom3` can also pass *template options* to the template. These options will only affect the template behaviour and are documented separately, in each template documentation.

7.2 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* to *value* if given, or `1` if *value* is omitted, 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.

`-N`
`--dump` Stop after the input file parsing stage, do not invoke any template and dump the parsed specification in dotgen format. This is mostly useful for debugging genom itself or to view the actual specification built by genom from a complex (set of) file(s). 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, one per line.

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 searched 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`.

Each component of *path* is searched for files matching `*/template.tcl`, where `*` is interpreted as the 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.

`-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.

7.3 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.

7.4 Environment variables

`GENOM_CPP` Define the C preprocessor program to use. The default is `libexec/genom-pcpp`. The `GENOM_CPP` program must recognize `-I` and `-D` arguments.

`PKG_CONFIG`
Define the path to the `pkg-config(1)` program.`pkg-config(1)` may be spawned by `genom-pcpp` for handling the `#pragma require` directive. The default is to search in the `PATH` variable.

`GENOM_TMPL_PATH`
The value of `GENOM_TMPL_PATH` is a colon-separated list of directories, much like `PATH`, where GenoM 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

Templates are the heart of the code generation system: they read a `dotgen` specification and produce a bunch of source files from it. Templates can produce virtually any kind of code, but they are usually used to produce a server or a client implementation of the component(s) described in the input specification.

A template is invoked on a `dotgen` specification by invoking GenoM with the template name followed by one or several input files (see Chapter 7 [Running], page 37 for details).

A number of base templates are provided by GenoM. These are the `skeleton`, `mappings` and `interactive` templates described hereafter. Additional templates can be made available by installing extra packages. At the time of writing, there exist for instance a `pocolibs` and a `ros` template that both provide a server and several kind of clients implementations. The list of available templates can be obtained by the command `genom3 -l` (see Section 7.2 [General options], page 37).

New templates can be developed by using the `TCL` code generator engine (see Section 8.4 [Creating Templates], page 43).

8.1 Creating initial codels skeleton

The skeleton template generates the skeleton of the codel functions defined in the input .gen file. It also generates a sample build infrastructure for building them. By default, files are generated in the same directory as the input .gen file. The `-C` option can be used to specify another output directory.

The `-l c++` option is specific to `C` codels. It generates a skeleton that compiles the codels with a `C++` compiler. This is useful for invoking `C++` code from the codels (Note that this is different from having `C++` codels.)

Files generated with this template are freely modifiable (and are actually required to be modified in order to provide some real codels). They are provided only as a sample - yet sensible - implementation. The only requirement is that codels provide a `pkg-config` file (`.pc`) named `<component>-genom.pc` and telling the other templates how to link with the codels library.

The template can also be invoked in *merge* mode, where it updates existing skeletons. This mode tries to merge modifications in the .gen file, for instance service addition or new interface definitions, into existing codels. In case of conflicting files, there are several merge strategies: option `-u` places conflicts markers in the source file, option `-i` interactively asks what to do, and the generic option `-m tool` runs `tool` on the conflicting files. `tool` can be any merge tool, for instance `meld`.

Example:

```
user@host:~$ genom3 skeleton demo.gen
creating ./codels/demo_motion_codels.c
creating ./codels/demo_codels.c
[...]
creating ./codels/Makefile.am
```

Supported options:

`-l c++`
`--language=c++`
: Compile C codels with a C++ compiler

`-C`
`--directory=dir`
: Output files in dir instead of source directory

`-m`
`--merge=tool`
: Merge conflicting files with tool

`-i`
: Interactively merge conflicting files, alias for -m interactive

`-u`
: Automatically merge conflicting files, alias for -m auto

`-f`
`--force`
: Overwrite existing files (use with caution)

`-h`
`--help`
: Print usage summary (this text)

8.2 Generating IDL mappings

This template generates a source file containing the native type definitions for all IDL types defined in the .gen input file. By default, types are generated for the codels language (defined in the .gen file) for the first available component. This can be changed with the -l option. The generated mappings are output on stdout.

Additionnaly, a dependency file suitable for inclusion in a `Makefile` can be generated. This is controlled by the `-MD`, `-MF` and `-MT` options. These options are documented herafter, and follow the same syntax as the same options of `gcc`.

Example:

```
user@host:~$ genom3 mappings demo.gen > demo_c_types.h
user@host:~$ genom3 mappings -l c++ demo.gen > demo_cxx_types.h
```

Supported options:

`-l`
`--language=lang`
: Generate mappings for language

`--signature`
: Generate codel signatures and types mappings

`-MD`
: Generate dependency information (in out.d)

`-MF=file`
: Generate dependency in file instead of out.d

`-MT=target`
: Change the target of the dependency rules

`-h`
`--help`
: Print usage summary (this text)

8.3 Running the TCL engine interactively

This template exports all the objects from the input .gen file for interactive use in a `tclsh` interpreter. The GenoM TCL engine procedures are available as in regular (scripted) templates.

This template is mostly useful for development of new templates or troubleshooting existing ones.

Example:

```
user@host:~$ genom3 interactive demo.gen
% foreach c [dotgen components] { puts [$c name] }
demo
% exit
user@host:~$
```

Supported options:

`-b` Batch mode: disable line editing facility

`-h`
`--help` Print usage summary (this text)

8.4 Creating new templates

8.4.1 The complete TCL engine reference

Source additional template code

`template require` *file* [TCL Backend]

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 (pxref{dotgen template dir}).

Parameters:

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

Generate template content

`template parse` [**args** *list*] [**perm** *mode*] [**file**|**string**|**raw** *file* ...] [TCL Backend]

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 GenoM 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 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 the text processed by the template engine. Finally, a raw, unprocessed source that is copied verbatim to the destination is specified as `raw` *text*, where *text* is the text to be output.

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 within the processed template source files from the usual *argv* variable.

Parameters:

args list 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.

perm mode This optional argument may be set to specify the permissions to be set for the created file.

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`.

Create symbolic links

`template link` *src dst* [TCL Backend]

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.

Define template options

`template options` *{ pattern body ... }* [TCL Backend]

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 dependencies

`template deps` [TCL Backend]

Return the comprehensive list of template files processed so far. This includes files processed via `template require`, `template parse` and `template link`. This list is typically used to generate dependency information in a Makefile.

Retrieve options passed to templates

`template arg` [TCL Backend]

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

Define template help string

`template usage` [*string*] [TCL Backend]

With a *string* argument, this procedure defines the template `"usage"` message. Unless the template redefines a `-h` option with `template options` (see [template options], page 44), 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.

Print runtime information

`template message` [*string*] [TCL Backend]

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

Abort template processing

`template fatal` [*string*] [TCL Backend]

Print an error message and stop. The message indicates the error location as reported by the `TCL` command [info frame].

Engine output configuration

`engine mode` [[+-]*modespec*]... [TCL Backend]

Configures various engine operating modes. `engine mode` 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).

Parameters:

modespec A mode specification string. If `mode` string is prefixed with a dash (`-`), it is turned off. If mode is prefixed with a plus (`+`) or not prefixed, it is turned on. Supported *modespec* are:

`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.

`merge-if-change` when turned on, existing destination files will be merged with new content by the engine, instead of being overwritten (see [engine merge-tool], page 46). `merge-if-change` is off by default.

`silent` when on, this mode avoids scattering standard output with informative messages from the code generator.

`debug` when on, this mode preserves temporary files and tcl programs generated in the temporary directory. Useful only for debugging the template.

Returns:

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

Example:

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

Automatic merge of generated content

`engine merge-tool` *tool* [TCL Backend]

Changes the engine merge tool. When the engine is in 'merge-if-change' mode (see see [engine mode], page 45), a merge tool is inkoked with the two conflicting versions of the destination file. If the merge tools exits successfuly, the generated file is replaced by the merged version.

There are two builtin tools: `interactive` and `auto`. `interactive` interactively prompts the user for each patch to be applied to merge the final destination. The user can accept or reject the patch, or leave the destination file unchanged. The `auto` builtin tool automatically merges the two files and places conflict markers (`<<<<<<<` and `>>>>>>>`) were appropriate in the destination file.

Parameters:

tool The path to the merge tool executable (e.g. `meld`), or one of the builtin keywords `interactive` or `auto`.

Change output directory

`engine chdir` *dir* [TCL Backend]

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.

Get current output directory

`engine pwd` [TCL Backend]

Returns:

The current engine output directory.

Genom program path and command line

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

`dotgen genom program` [TCL Backend]

Return the absolute path to the GenoM executable currently running.

`dotgen genom cmdline` [TCL Backend]

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

`dotgen genom version` [TCL Backend]

Returns the full version string of the GenoM program.

`dotgen genom templates` [TCL Backend]

Return the list of all currently available templates name.

`dotgen genom debug` [TCL Backend]

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

`dotgen genom verbose]` [TCL Backend]

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

Template path and directories

Those commands return information about the template currently being parsed.

`dotgen template name` [TCL Backend]

Return the current template name.

`dotgen template dir` [TCL Backend]

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

`dotgen template builtindir` [TCL Backend]

Return a path to the genom builtin templates source directory.

`dotgen template tmpdir` [TCL Backend]

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

Input file name and path

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

`dotgen input notice` [TCL Backend]

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.

`dotgen input deps` [TCL Backend]

Return the comprehensive list of input files processed so far. This includes the input `.gen` file itself, plus any other file required, directly or indirectly, via a `#include` directive. This list is typically used to generate dependency information in a Makefile.

Process additional input

`dotgen parse` **file**|**string** *data* [TCL Backend]

Parse additional `.gen` data either from a file or from a string. When parsing is successful, the corresponding objects are exported to the backend.

Parameters:

file|string Specify if parsing from a file or from a string.

data When parsing from a file, data is the file name. When parsing from a string, data is the string to be parsed.

Data type definitions from the specification

`dotgen types [pattern]` [TCL Backend]

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 the type names with *pattern*. 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 of class `type`.

Components definitions from the specification

`dotgen components [pattern]` [TCL Backend]

This command returns the list of components 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 component command that can be used to access detailed information about each particular component object.

Parameters:

pattern Filter 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 of class `component`.

Interfaces definitions from the specification

`dotgen interfaces [pattern]` [TCL Backend]

This command returns the list of interfaces 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 an interface command that can be used to access detailed information about each particular interface object.

Parameters:

pattern Filter the interface 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 interface objects of class `interface`.

Target programming language

`lang` *language* [TCL Backend]

Set the current language for procedures that output a language dependent string

Parameters:

language The language name. Must be one of `c` or `c++`.

Generate comment strings

`comment` [*-c*] *text* [TCL Backend]

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

Parameters:

c The string to use as a comment character (overriding current language).

test The string to be commented.

Cannonical file extension

`fileext` [*-kind*] [TCL Backend]

Return the cannonical file extension for the current language.

Parameters:

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

Generate indented text

`indent` [*#n*|`++`|`--`] [*text* ...] [TCL Backend]

Output *text*, indented to the current indent level. Each *text* argument is followed by a newline. Indent level can be changed by passing an absolute level with *#n*, or incremented or decremented with `++` or `--`.

Parameters:

test The string to output indented.

Generate filler string

`---` [*-column*] *text ... filler* [TCL Backend]

This command, spelled with 3 dashes (`-`), return a string of length *column* (70 by default), starting with *text* and filled with the last character of the *filler* string.

Parameters:

text The text to fill.

filler The filler character.

column The desired length of the returned string.

Chop blocks of text

`wrap [-column] text [prefix] [sep]` [TCL Backend]

Chop a string into lines of length *column* (70 by default), prefixed with *prefix* (empty by default). The string is split at spaces by default, or at *sep* if given.

Parameters:

text	The text to fill.
prefix	A string prefixed to each line.
sep	The separator for breaking text.
column	The desired maximum length of each line

Cannonical object name

`cname string|object` [TCL Backend]

Return the cannonical name of the *string* or the GenoM *object*, according to the current language.

If a regular string is given, this procedure typically maps IDL :: scope separator into the native scope separator symbol for the current language. If a codel object is given, this procedure returns the symbol name of the codel for the current language.

Parameters:

string	The name to convert.
object	A GenoM object.

Unique type name

`language mangle type` [TCL Backend]

Return a string containing a universally unique representation of the name of the *type* object.

Parameters:

type	A 'type' object.

IDL type language mapping

`language mapping [type]` [TCL Backend]

Generate and return a string containing the mapping of *type* for the current language, or of all types if no argument is given. The returned string is a valid source code for the language.

Parameters:

type	A 'type' object.

Code for type declarations

`language declarator type [var]` [TCL Backend]

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

Parameters:

type	A 'type' object.
var	A string representing the name of a variable of type *type*.

Code for variable addresses

`language address` *type* [*var*] [TCL Backend]

Return an expression evaluating to the address of a variable in the current language.

Parameters:

type	A 'type' object.
var	A string representing the name of a variable of type *type*.

Code for dereferencing variables

`language dereference` *type kind* [*var*] [TCL Backend]

Return an expression dereferencing a parameter passed by value or reference, in the current language.

Parameters:

type	A 'type' object.
kind	Must be `value` or `reference`.
var	A string representing the name of a parameter of type *type*.

Code for declaring functions arguments

`language argument` *type kind* [*var*] [TCL Backend]

Return an expression that declares a parameter *var* of type *type*, passed by value or reference according to *kind*.

Parameters:

type	A 'type' object.
kind	Must be `value` or `reference`.
var	A string representing the name of a variable of type *type*.

Code for passing functions arguments

`language pass` *type kind* [*var*] [TCL Backend]

Return an expression that passes *var* of type *type* as a parameter, by value or reference according to *kind*.

Parameters:

type	A 'type' object.
kind	Must be `value` or `reference`.
var	A string representing the name of a variable of type *type*.

Code for accessing structure members

`language member` *type mlist* [TCL Backend]

Return the language construction to access a member of a *type*. *mlist* is a list 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`).

Parameters:

type A 'type' object.

mlist A list of hierachical members to access.

Code for declaring codel signatures

`language signature` *codel* [*separator*] [*location*] [TCL Backend]

Return the signature of a codel in the current 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:

code A 'codel' object.

separator A string, inserted between the return type and the codel symbol name.

location A boolean indicating whether to generate `#line` directives corresponding to the codel location in `.gen` file.

Code for calling codels

`language invoke` *codel params* [TCL Backend]

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

Parameters:

code A 'codel' object.

params The list of parameters passed to the codel. Each element of this list must be a valid string in the current language corresponding to each parameter value or reference to be passed to the codel (see [language pass], page 51).

IDL Type manipulation procedures

Those commands manipulate IDL type objects and return information about them. They all take a type object as their first argument, noted $type in the following command descriptions. Such an object is typically returned by other procedures, such as `dotgen types` (see [dotgen types], page 48).

`$type kind` [TCL Backend]

Return a string describing the nature of the IDL type, such as `long`, `double`, `struct` ...

`$type name` [TCL Backend]

Return the name of the IDL type. No namespace components are included in the result.

`$type fullname` [TCL Backend]

Return the fully qualified name of the IDL type. The result includes the namespace hierarchy in which the type is defined and the last component is the result of `$type name`.

`$type scope` [TCL Backend]

Return the list of lexical scopes in which the type is defined. Each element of the list contains two values: the nature of the scope and its name. The nature of the scope will be either `module` for IDL modules, or `struct` if the type is defined inside an IDL `struct`.

`$type fixed` [TCL Backend]

Return a boolean indicating if the type is of fixed, constant size (true) or not (false).

`$type final` [TCL Backend]

Return a new type object with all aliases resolved. For `const` types, this returns the type of the constant. For `typedef`, this return the first non-aliased type. For `struct` and `union` members, this returns the type of the member.

`$type parent` [TCL Backend]

Return the parent type of a nested type definition, or raise an error if the type is not nested.

`$type nested` [TCL Backend]

Return the nested types defined by the given type

`$type types [`*`filter`*`]` [TCL Backend]

Return the list of all types that are recursively used by the given type. In other words, this is the list of types that must be known in order to completely define the given $type.

For all basic types such as `long`, `double` and so forth, the returned list is empty. For `enum`erated types, the result is the list of enumerators. For aggregates such as `struct`, `union` or `exception`, the result is the list of all members, expanded recursively with the same rules. Finally, for `array`s or `sequence`s, this returns the type of the array or sequence elements, recursively expanded.

Parameters:

filter The optional filter can be used to filter out some elements from the type list. The filter must be a tcl anonymous function (see tcl [apply] command) that accepts one argument that is a genom object. It must return a boolean to indicate whether the type should be included (true) or excluded (false).

`$type type` [TCL Backend]

Return the underlying type of a type that contains another type definition. For instance, this procedure invoked on an `array` or `sequence` type returns the element type. It returns the aliased type for `typedef`.

`$type length` [TCL Backend]

Return the length of an `array`, `sequence` or `string` type.

`$type value` [TCL Backend]

Return the value associated with a `const` type.

`$type valuekind` [TCL Backend]

Return the nature of the value associated with a `const` type.

`$type members` [TCL Backend]

Return a list of types defined by the given `struct`, `union` or `enum` type.

`$type discriminator` [TCL Backend]

Return the discriminator of the given `union`.

`$type port` [TCL Backend]

Return the port object referenced by the given `port`.

`$type remote` [TCL Backend]

Return the remote object referenced by the given `remote`.

`$type cname` [TCL Backend]

Return a string representing the type name in the current language.

`$type mangle` [TCL Backend]

Return a string uniquely describing the given type, suitable for use as an identifier in source code written in the current programming language.

`$type mapping` [TCL Backend]

Return an ASCII string representing the implementation (definition) of the given type, suitable for use in source code written in the current programming language.

`$type declarator` [*var*] [TCL Backend]

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

Parameters:

var The variable being declared. If not given, an abstract declarator is returned.

`$type address` *var* [TCL Backend]

Return an expression representing the address of a variable of the given type in the current language.

Parameters:

var The variable of which the address must be taken.

`$type argument` **value|reference** [*var*] [TCL Backend]

Return an expression that declares a parameter *var* of the given type, passed by value or reference according to the second parameter.

Parameters:

var The argument name being declared. If not given, an abstract declarator is returned.

`$type pass` **value|reference** *var* [TCL Backend]

Return an expression that passes a variable *var* of the given type as a function parameter. The variable is passed by value or reference according to second argument.

Parameters:

var The variable that must be passed.

`$type dereference` **value|reference** *var* [TCL Backend]

Return an expression that retrieves the value of a parameter *var*, passed by value or reference according to the second argument.

Parameters:

var The argument name.

`$type digest` [TCL Backend]

Return an ASCII representaion (32 characters) of an MD5 digest of the given type. This is useful for implementing a cheap runtime verification that two types match.

`$type masquerade` [TCL Backend]

Return any value defined in a `#pragma masquerade` for that type, if the current template matches the corresponding parameter of the `#pragma`. See Section 5.17.3 [#pragma masquerade], page 17.

`$type loc` [TCL Backend]

Return list describing the source location where that type is defined. The list contains three elements: the file name, the line number and the column number.

`$type class` [TCL Backend]

Always returns the string `"type"`. Useful to determine at runtime that the object is a type object.

Indices

Index of concepts

Index of TCL backend procedures

$

-

dotgen genom verbose

C

D

E

F

I

L

T

W