Bosal — Test Harness Generator

Automatic Test Harness Generation for Micca Domains

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	April 10, 2017	Initial coding started.	GAM
0.2	September 2, 2017	Changed approach to use ability of rosea to restore a micca population.	GAM
1.0	September 6, 2017	First release.	GAM
1.0.1	September 10, 2017	Correct problem in formatting state machine trace responses.	GAM
1.1	September 21, 2017	Correct problem in keeping track of the state of tracing. Added instance ids to instance query response. Separated responses into different contexts to prevent interference. Added timestamps to instrumentation traces.	GAM
1.2	September 22, 2017	Rational fatal error message traces to be of the same form as instrumentation traces.	GAM

Contents

List of Figures

Harnessing micca generated domains

This document is about a program named `bosal`. This document is also a literate program and contains all the design information and code for the `bosal` program. Readers unfamiliar with literate programs should consult the appendix for more details.

Bosal harness concepts

When `micca` is used to translate an xUML domain into code, the resulting "C" file can be combined with a `main` function and any necessary external entity stubs[1] to build an executable program for the domain. The resulting executable, unfortunately, does not do very much. A domain in isolation, without being bridged to other service domains, doesn't have any way of being triggered to execute. We need the ability to invoke domain operations, signal events, and perform other model level actions on the domain and to observe its execution. We would like to do all this in a controlled environment that does not necessarily include the entire set of domains in an application.

The `micca` run-time is capable of executing in a POSIX environment. This is intended primarily to simulate and integrate a system that is ultimately targeted for a micro-controller. A POSIX environment has more capability for testing and introspection and a substantial amount of the integration and testing for an application can be accomplished there. To support this important activity, `bosal` is used to automatically generate a test harness in which one or more domains may be run in a POSIX environment under the control of the harness.

Test harness structure

The intended workflow is for the set of domains for an application to be tested individually and then integrated together, adding one additional domain at a time, until the complete application can be tested together.

The following figure shows schematically how a `bosal` generated harness is structured.

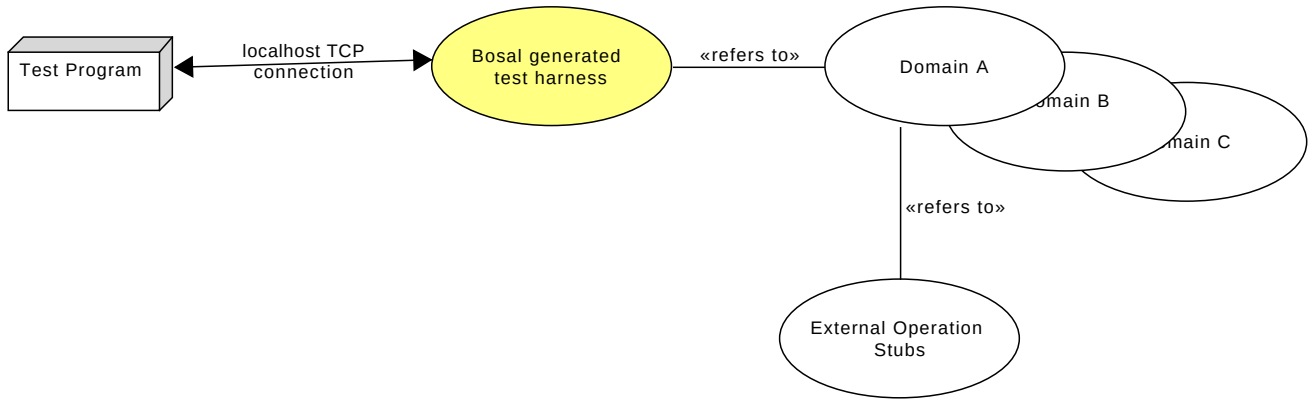

Figure 1: Test Harness Structure Overview

The `bosal` generated test harness contains code that creates a localhost TCP/IP server. Communication is across an ordinary TCP socket. The protocol is ASCII text, record oriented, and formulated as commands. It is possible, although not very convenient, to interact with a test harness using `telnet(1)`.

A test harness can be generated that contains multiple domains. Any external operations of the harnessed domains must be resolved. This can be done using separate code files or by requesting micca to generate the stubs. In the multiple domain case, usually there are bridge code files that take the place of some of the external operations stubs. By compiling and linking the `bosal` generated harness code, the `micca` generated domain code, and any other required bridges or stubs, a program is obtained that can control the domain execution via commands received across the communications interface.

[1] `micca` can also generate the external entity stubs

By reading the files saved by `micca`, `bosal` can generate the code that maps the command protocol onto the specifics of a domain. For example, domain operations can be invoked using the `domainop` command. This command takes arguments of which domain the operation refers to, the name of the operation, and any arguments the operation requires. The test harness code then arranges to invoke the "C" function that is the named domain operation.

Building a test harness

To complete building a test harness, a `main` function must be provided. The `main` function must invoke `mrt_Initialize()` before any domain or harness functions are invoked. The test harness is started by invoking `bsl_Run()`. Other initialization may be done in `main`. For example, interacting with other systems or programs or invoking domain initialization functions are candidates for placing in `main`. The code below shows a minimal `main` function necessary to run a test harness. Upon request, `bosal` will generate a minimal `main` function directly into the harness code file.

```
#include "bosal.h"
#include "micca_rt.h"

int
main(
    int argc,
    char **argv)
{
    /*
     * Initializing the micca run time must be done before any other
     * domain or harness functions are invoked.
     */
    mrt_Initialize() ;

    /*
     * It may be necessary to initialize domains or bridges. This can
     * be specific to the needs of a particular situation.
     */

    /*
     * Finally, the harness is set to run. The harness code will take
     * over handling the micca event loop.
     */
    bsl_Run() ;
}
```

Invoking `bosal`

`bosal` *options savefile1 savefile2 ...*

where options are:

-version
Print out version information and exit.

-header
Output a copy of the header file, `bosal.h`, and exit.

-doc
Output a copy of the manual documentation and exit.

-output ***filename***
Name of the output file. If this option is not supplied, then `bosal` writes the test harness code to the standard output.

-main
include a minimal `main` function in harness output.

-nogenerate
Don't generate any output.

-level ***loglevel***
Logger print level. If this option is not supplied, then the logging level is set to `warn`.

savefileN
The names of the files, saved by `micca`, for the domains that are to be configured into the test harness.

Harness communications protocol

Once a test harness is executing, clients can connect to TCP port 3906, by default, on localhost to access the test harness. Only one client may access a test harness. Any further connection attempts are refused. After a client has connected, terminating the connection causes the test harness to terminate.

The protocol accepted on the communications socket of the test harness is ASCII record oriented. Each command is sent as a sequence of whitespace separated words terminated by `CR/LF`. Words that contain embedded whitespace must be quoted by enclosing them in braces (`{}`). Words that contain an unmatched set of braces must precede the unmatched brace character by a backslash (`\`), *e.g.* `{\{}`.

Responses from the test harness are of four types:

Command responses
are of the form `cmd {<command response>}`.

Fatal run-time errors
are of the form `fatal {<fatal error record>}`.

State machine traces
are of the form `trace {<state trace record>}`.

Execution instrumentation
are of the form `instr {<instrumentation record>}`.

Commands sent to the test harness always receive a response and client should not send another command until a response has been received. However, if tracing or instrumentation is turned on, state machine trace records and execution instrumentation are sent asynchronously to any command / response sequence. Also, any fatal errors are sent asynchronously. Client connections to the test harness must be prepared to receive trace and execution records if they enable these functions. At start up, tracing and instrumentation are tuned off and initial interaction with a test harness is strictly command / response in nature with the possibility of a fatal error.

Command responses

A command response consists of two words: `cmd` followed by a `<command response>` word. The syntax of a `<command response>` is a set of whitespace separated words that form name / value pairs. A command response consists of three name / value pairs:

`name <command name>`
The name of the command is returned in its response.

`result <command execution result>`
The result of the command is returned. The format of the `<command execution result>` depends upon the nature of the command and is described below for each command.

`status <command execution status>`
The execution status of the command is returned. If `<command execution status>` is the string, **ok**, then the command executed successfully. If it is, **error**, then the command failed and the `result` value is a human readable error message.

Fatal errors

A fatal error record consists of two words: `fatal` followed by a `<fatal error detail>` word. Fatal errors indicate a *panic* condition in the execution of the domain. The `<fatal error detail>` consists of the following set of name / value pairs.

`time <time stamp>`
A timestamp of when the fatal error was generated.

`message <error text>`
A human readable text message produced by the `micca` run-time. The text has no fixed, defined format.

Normally, such panic conditions would cause the program to terminate. By default the `micca` run-time code invokes the standard library function, `abort()`, when a fatal error occurs. However, a test harness installs its own fatal error handler to insure it does not terminate and the communications interface continues to operate. This allows additional queries to attempt to determine the cause of the error or to inquire as the to state of other components.

State trace records

When tracing in turned on, the test harness will send a `trace` record each time an event is dispatched to a class instance or an assigner. A trace response consists of two words: `trace` followed by a `<state trace record>` word. The state trace record, simular to a command response, consists of a set of name / value pairs.

`type <transition type>`
The type of the transition. `<transition type>` will be one of the following strings.

- **transition**
- **polymorphic**
- **creation**

The **type** of the transition determines the set of name / value pairs that included in the trace record. For all trace records, the following is included in the trace record.

`event <event name>`
The name of the event.

`source <event signaler>`
The class instance that signaled the event.

target <event recipient>
The class instance that received the event.

time <event dispatch time>
A timestamp of when the dispatch occurred.

For **transition** type traces, the trace record additionally contains:

currstate <recipient current state>
The current state of the receiving class instance before the event transition takes place.

newstate <recipient new state>
The new state of the receiving class instance after the transition has happened.

For **polymorphic** type traces, the trace record additionally contains:

relationship <relationship name>
The name of the generalization across which the polymorphic event is signaled.

newevent <event name>
The name of the polymorphic event when mapped to an event for the target class.

subclass <class name>
The name of the subclass that is to receive the polymorphic event. The is the subclass to which the superclass instance is related at the time the polymorphic event is dispatched.

For **creation** type traces, no additional information is provided.

Instrumentation records

Instrumentation records contain traces of functions that are executed. `Micca` will generate, by default, execution traces if the `MRT_INSTRUMENT` preprocessor symbol is defined when the domain code is compiled. If the preprocessor symbol `BOSAL` also is defined when the domain is compiled, the automatically generated instrumentation output will be sent on the communications interface as an instrumentation response. An instrumentation response consists of two words: `instr` followed by a `<instrumentation detail>` word. An instrumentation detail consists of the following set of name / value pairs.

time <time stamp>
A timestamp of when the instrumentation record was generated.

message <instrumentation text>
A text message produced by the running domain. Some instrumentation message are automatically inserted when the domain is generated by `micca`. Others may be added by the programmer using the `MRT_DEBUG` macro. Consequently, the instrumentation text has no fixed format.

Command protocol details

The following sections describe each command accepted by a test harness. The commands are listed in alphabetical order. The → symbol is used to indicate the text sent to the test harness. The ← symbol is used to indicate the response. Note that only successful responses are shown below. Any error occurring during execution of the command results in the response containing the `code error` field and the result is a diagnostic error string.

Cancel command

The `cancel` command cancels a delayed signal.

→	`cancel` *domain class inst event*
←	`cmd {name cancel result {} status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst	is the name or number of an instance. Pre-existing instances have names given by the domain population. All instances can be referred to by an instance number.
event	is the name of an event in the state model of the *class*.

Create command

The `create` command creates a class instance. Note that creating an instance in a test harness often result in a violation of referential integrity since most classes participate in unconditional relationships and it is not possible to relate instances together using the test harness commands.

→	`create` *domain class*
←	`cmd {name delete result` *inst_number* `status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst_number	is the number of the class instance that was created. This number can be used to refer to the instance in other commands.

Createasync command

The `createasync` command sends a creation event.

→	`createasync` *domain class event* ? *arg1 arg2 ...* ?
←	`cmd {name createasync result` *inst_id* `status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*. *event* is the name of an event in the state model of the *class*.
argN	arguments to the event. The order of the arguments must be the same as the order given in the state model definition.
inst_id	is the number of an instance. The newly created instance can be referred to by this number in other commands.

Delaysignal command

The `delaysignal` command sends a signal at a time in the future.

→	`delaysignal` *domain class inst delay event* ? *arg1 arg2 ...* ?
←	`cmd {name signal result {} status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst	is the name or number of an instance. Pre-existing instances have names given by the domain population. All instances can be referred to by an instance number.
delay	is the minimum number of milliseconds that will elapse before the signal is delivered. A *delay* value of 0 implies that the event is delivered immediately.
event	is the name of an event in the state model of the *class*.
argN	arguments to the event. The order of the arguments must be the same as the order given in the state model definition.

Delete command

The `delete` command deletes a class instance.

→	`delete` *domain class inst*
←	`cmd {name delete result {} status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst	is the name or number of the class instance to delete.

Domainop command

The `domainop` command executes a domain operation.

→	`domainop` *domain operation* ? *arg1 arg2 ...* ?
←	`cmd {name domainop result` *operation_result* `status ok}`
domain	is the name of a domain.
operation	the name of an operation in the domain. The operation name is the same as that given in the domain definition.
argN	arguments to the domain operation. The order of the arguments must be the same as the order given in the domain definition.
operation_result	The result of the domain operation is dependent upon the return value of the domain operation.

Event loop command

The `eloop` command gives control over event dispatching and how the test harness is allowed to execute. It is possible to halt event dispatch and also to dispatch events one by one or as an entire thread of control.

→	`eloop` ? *operation* ?
←	`cmd {name eloop result` *eloop result* `status ok}`
operation	event loop operation. The *operation* argument is optional. If present, it must be one of the following strings: • **run** ⇒ run the event loop to dispatch events. • **halt** ⇒ halt the event loop. No further events are dispatched. • **once** ⇒ dispatch one event from the event loop. • **toc** ⇒ dispatch one thread of control from the event loop. The **toc** operation can be specified optionally with either a **wait** or **nowait** argument. If **wait** is specified (or by default if no **toc** option is specified), then the event loop will wait until a thread of control event arrives. If **nowait** is specified, the command will return immediately if no thread of control is ready to dispatch. This option is useful if the event that starts the thread of control is issued via a delayed signal.
eloop result	is the current run state of the event loop, either **running** or **halted**, if the *operation* argument is missing, or if the *operation* argument is given as **run** or **halt**. The *eloop result* is either **true** or **false** if the *operation* is **once** or **toc** and indicates if any event or thread of control was actually dispatched. It is an error to attempt to execute the **once** or **toc** operations if the event loop is currently **running**.

Null command

The `null` command performs no action, successfully. It can be used to insure that the test harness is running and responsive.

→	`null`
←	`cmd {name null result {} status ok}`

Instrument command

The `instr` command controls whether instrumentation records are send on the communications connection. Clients may use it to change the amount of execution trace instrumentation coming from the test harness.

→	`instr` ? *on* | *off* ?
←	`cmd {name instr result` *instr_result* `status ok}`
instr_result	The current state of the instrumentation record output, either **on** or **off**.

Query command

The `query` command returns information about the model components in the test harness.

→	`query` *query_subcommand subcommand_args*
←	`cmd {name query result` *query_result* `status ok}`

The `query` command has subcommands to retrieve information about specific model components. The following sections describe each subcommand and its response.

Query domains

The `query domains` subcommand obtains the domain names in test harness.

→	`query domains`
←	`cmd {name query result` *domain_list* `status ok}`
domain_list	a series of whitespace separated domain names that included in the test harness.

Query domain operations

The `query operations` subcommand obtains the names of the domain operations for a given domain.

→	`query operations` *domain*
←	`cmd {name query result` *op_list* `status ok}`
op_list	a series of whitespace separated domain operation names for the given *domain*.

Query domain operation parameters

The `query doparams` subcommand obtains the names and data types of the parameters for the given domain operation.

→	`query doparams` *domain operation*
←	`cmd {name query result` *param_list* `status ok}`
param_list	a series of whitespace separated parameter name / parameter data type word pairs. The length of *param_list* is always even with the even index words giving a parameter name and the odd indexed word giving its corresponding data type. The length of *param_list* is 0, if the operation requires no parameters.

Query classes

The `query classes` subcommand obtains the names of the classes in a domain.

→	`query classes` *domain*
←	`cmd {name query result` *class_list* `status ok}`
domain	the name of a domain in the test harness.
class_list	a series of whitespace separated class names that are part of *domain*.

Query attributes

The `query attributes` subcommand obtains the names of the attributes of a class.

→	`query attributes` *domain class*
←	`cmd {name query result` *attribute_list* `status ok}`
domain	the name of a domain in the test harness.
class	the name of a class in *domain* .
attribute_list	a series of whitespace separated attribute names that are part of *class*.

Note that only attributes that may be read and updated are returned.

Query instances

The `query instances` subcommand obtains the number and names of the instances of a class.

→	`query instances` *domain class*
←	`cmd {name query result` *instance_result* `status ok}`
domain	the name of a domain in the test harness.
class	the name of a class in *domain* .
instance_result	a set of two name / value pairs of the form • **total** *instance_count* **instance_count** is the maximum number of instances of *class* that can exist. • **named** *instance_id_pairs* **instance_id_pairs** a list consisting of an even number of elements. The even elements are the names given to initial instances and the odd elements are the unique numerical identifiers of the instance within the class. Instances that are part of the initial instance population have names and are returned in this list.

Query states

The `query states` subcommand obtains the names of the states in the state model of a class.

→	`query states` *domain class*
←	`cmd {name query result` *state_list* `status ok}`
domain	the name of a domain in the test harness.
class	the name of a class in *domain* .
state_list	a series of whitespace separated state names that are in the state model of *class*.

Note it is an error to query the states of a class that has no state model.

Query events

The `query events` subcommand obtains the names of the events in the state model of a class.

→	`query events` *domain class*
←	`cmd {name query result` *event_list* `status ok}`
domain	the name of a domain in the test harness.
class	the name of a class in *domain* .
event_list	a series of whitespace separated event names that are in the state model of *class*.

Note it is an error to query the events of a class that has no state model.

Query event parameters

The `query evarams` subcommand obtains the names and data types of the supplimental parameters for the given class event.

→	`query evparams` *domain class event*
←	`cmd {name query result` *param_list* `status ok}`
param_list	a series of whitespace separated event parameter name / event parameter data type word pairs. The length of *param_list* is always even with the even index words giving a parameter name and the odd indexed word giving its corresponding data type. The length of *param_list* is 0, if the event requires no parameters.

Query current

The `query current` subcommand obtains the name of the current state of an instance of a class.

→	`query current` *domain class instance*
←	`cmd {name query result` *current_state* `status ok}`
domain	the name of a domain in the test harness.
class	the name of a class in *domain* .
inst	is the name or number of an instance of *class*.
current_state	the name of the current state of *instance*.

Note it is an error to query the current state of a class that has no state model.

Read command

The `read` command is used to obtain the values of one or more attributes.

→	`read` *domain class inst* ? `_attr1 attr2 ...` ?
←	`cmd {name read result` *value_result* `status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst	is the name or number of an instance of *class*.
attrN	an optional series of names of attributes of *class*. If no *attrN* arguments are given, then the values of all the attributes of the instance are returned. Otherwise, the values of the given *attrN* attributes are returned.
value_result	a series of white space separated attribute name / attribute value pairs.

Remaining command

The `remaining` command obtains the number of milliseconds remaining before a delayed event is dispatched.

→	`remaining` *domain class inst event*
←	`cmd {name cancel result` *remaining_time* `status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst	is the name or number of an instance. Pre-existing instances have names given by the domain population. All instances can be referred to by an instance number.
event	is the name of an event in the state model of the *class*.
remaining_time	the number of milliseconds in the future when *event* is to be dispatched. A *remaining_time* of 0 implies that the event has already been dispatched or was never signaled.

Signal command

The `signal` command signals an event to a class instance.

→	`signal` *domain class inst event* ? *arg1 arg2 ...* ?
←	`cmd {name signal result {} status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst	is the name or number of an instance. Pre-existing instances have names given by the domain population. All instances can be referred to by an instance number. *event* is the name of an event in the state model of the *class*.
argN	arguments to the event. The order of the arguments must be the same as the order given in the state model definition.

Trace command

The `trace` command controls whether event dispatch trace records are send on the communications connection.

→	`trace` ? *on* | *off* ?
←	`cmd {name trace result` *trace_result* `status ok}`
trace_result	The current state of the trace record output, either **on** or **off**.

Update command

The `update` command is used to update the values of one or more attributes.

→	`update` *domain class inst* ? `_attr1 value1 attr2 value2 ...` ?
←	`cmd {name read result` *value_result* `status ok}`
domain	is the name of a domain.
class	is the name of a class within *domain*.
inst	is the name or number of an instance of *class*.
attrN valueN	an optional series of attribute name / attribute value pairs. If no *attrN valueN* arguments are given, then no updates occur. Otherwise, the values of the given *attrN* attributes are updated.
value_result	a series of white space separated attribute name / attribute value pairs reflecting the current values of the attributes.

Version command

The `version` command performs returns the version number of `bosal` used to create the test harness code.

→	`version`
←	`cmd {name version result` *version_number* `status ok}`
version_number	the version `bosal` that created the test harness.

Domain independent code

Much of the test harness code is independent of the specifics of any domain in the harness. One alternative design is to place the domain independent code in a separate file and compile that code into a library. Here, we have decided to put both domain independent and domain dependent code into the same file. This design reduces any dependency between an external library code file and different versions of `bosal`. Since the amount of domain independent code is relatively small, removing external dependencies was deemed beneficial.

External harness functions

Running the harness

Normally, systems built from `micca` supply a `main()` function which invokes the `mrt_EventLoop()` function in the micca run-time to cause the domain to execute. This function manages all the state machine dispatch and execution sequencing. When running in a test harness, we want more control over the execution sequencing and must perform other initialization before allowing the domains to run. Consequently, test harness code is run by invoking `bsl_Run()`.

```
<<harness external function declarations>>=
extern void bsl_Run(void) ;
```

Invoking `bsl_Run()` initializes and runs a `bosal` generated test harness. Before invoking this function, it is necessary to invoke `mrt_Initialize()`.

The implementation of `bsl_Run` is as you might expect, an infinte loop that replaces the micca run-time event loop.

```
<<harness external function definitions>>=
void
bsl_Run(void)
{
    if (!bslInitialize()) {                                    // ❶
        return ;
    }

    mrt_SetFatalErrorHandler(bslErrorHandler) ;                // ❷

    for (bslHarnessState = bslRunning ; ;) {
        if (setjmp(bslFatalBuf) == 0) {                        // ❸
            mrt_EventLoop() ;                                  // ❹
        }

        for (bslHarnessState = bslHalted ; bslHarnessState == bslHalted ;) {
            bslReadAndEvalCommand(0) ;                         // ❺
```

```
        }
    }
}
```

1. First, we must initialize the test harness infrastructure.
2. The default error handler for the `micca` run-time will invoke `abort()` when an error occurs. This would cause the test harness to stop. We want more control so we can see transitions and probe the cause of an error.
3. The test harness error handler uses `setjmp/longjmp` to transfer control back to here if a fatal error occurs.
4. It is still `mrt_EventLoop` that causes everything to run. `mrt_EventLoop` will return if any action in the domain invokes `mrt_SyncToEventLoop` or if the harness error handler is invoked as a result of a fatal error. In either case, we consider the state of the event loop as halted.
5. Once the event loop is halted, we still need to read and evaluate test harness commands off the communications link so that an external program can interact with the harness.

The harness keeps track of whether or not it is in the `micca` run-time event loop and uses a variable to track that state.

```
<<harness static data>>=
static enum {
    bslHalted,
    bslRunning,
} bslHarnessState = bslHalted ;
```

Harness instrumentation

The test harness provides functions, modeled after `printf(3)`, for unstructured instrumentation output. These functions are usually used by instrumentation that is generated by `micca` and result in the output being formulated as an instrumentation trace on the harness communication socket.

```
<<harness external function declarations>>=
extern int
bsl_Vprintf(
    char const *format,
    va_list args) ;
```

format
: a `printf`-style format string describing the output.

args
: a variable argument list supplying parameters according to `format`.

```
<<harness external function definitions>>=
int
bsl_Vprintf(
    char const *format,
    va_list args)
{
    if (!instrOn || bslCmdStream == NULL) {
        return 0 ;
    }

    bslBeginResponse(InstrContext) ;
    bslAddWordToContext(InstrContext, "instr") ;
```

```
    bslPushContext(InstrContext) ;

    bslFormatToContext(InstrContext, "time %s message ", bslTimestamp()) ;

    bslPushContext(InstrContext) ;
    bslVformatToContext(InstrContext, format, args) ;

    return bslEndResponse(InstrContext) ;
}
```

```
<<harness external function declarations>>=
extern int
bsl_Printf(
    char const *format,
    ...) ;
```

format
A `printf`-style format string.

...
A variable number of function arguments matching the `format` specification.

```
<<harness external function definitions>>=
int
bsl_Printf(
    char const *format,
    ...)
{
    va_list args ;

    va_start(args, format) ;
    int n = bsl_Vprintf(format, args) ;
    va_end(args) ;

    return n ;
}
```

Handling fatal system errors

The `micca` run-time code diagnoses several fatal system errors. These are "panic" conditions and, by default, invokes `abort()` to terminate the process. We need to do better than that in a test harness since we might want to inquire and probe to determine the circumstances of the failure. Fortunately, we can install our own error handler to prevent the harness from terminating. The error handler arguments are dictated by the `micca` run-time.

```
<<harness forward function declarations>>=
static void bslErrorHandler(MRT_ErrorCode errNum, char const *fmt,
        va_list alist) ;
```

```
<<harness static function definitions>>=
static void
bslErrorHandler(
    MRT_ErrorCode errNum,
    char const *fmt,
    va_list alist)
{
    bslBeginResponse(FatalContext) ;
    bslAddWordToContext(FatalContext, "fatal") ;
```

```
    bslPushContext(FatalContext) ;
    bslFormatToContext(FatalContext, "time %s message ", bslTimestamp()) ;

    bslPushContext(FatalContext) ;
    bslVformatToContext(FatalContext, fmt, alist) ;

    bslEndResponse(FatalContext) ;

    longjmp(bslFatalBuf, 1) ;                          // ❶
}
```

❶ This transfers control back to `bsl_Run`.

The `setjmp/longjmp` combination that performs the non-local goto needs a variable to hold the context.

```
<<harness static data>>=
static jmp_buf bslFatalBuf ;
```

Harness initialization

The `bslInitialize` function sets up the communications and command infrastructure for the test harness. Most of the initialization work is involved in setting up the TCP / IP communications socket.

```
<<harness forward function declarations>>=
static bool bslInitialize(void) ;
```

```
<<harness static function definitions>>=
static bool
bslInitialize(void)
{
    <<bslInitialize: create socket>>
    <<bslInitialize: bind socket>>
    <<bslInitialize: listen on socket>>
    <<bslInitialize: register signals>>

    mrt_RegisterTraceHandler(NULL) ; // ❶

    return true ;
}
```

❶ Setting the trace handler to `NULL` disables the tracing.

Setting up the communications socket is standard POSIX systems programming.

```
<<bslInitialize: create socket>>=
bslServiceSock = socket(AF_INET, SOCK_STREAM, 0) ;
if (bslServiceSock == -1) {
    perror("unable to obtain AF_INET stream socket") ;
    return false ;
}
```

The socket number is held as file static data so it may be used for other communications purposes.

```
<<harness static data>>=
static int bslServiceSock = -1 ;
```

The socket will be used in a server role to listen for connections, so we must bind to a port that clients will know.

```
<<bslInitialize: bind socket>>=
struct sockaddr_in hsrvAddr ;
memset(&hsrvAddr, 0, sizeof(hsrvAddr)) ;
#       ifdef __APPLE__
hsrvAddr.sin_len = sizeof(hsrvAddr) ;
#       endif /* __APPLE__ */
hsrvAddr.sin_family = AF_INET ;
hsrvAddr.sin_port = htons(HARNESS_PORT) ;

static char const host[] = "localhost" ;            // ❶
struct hostent *hostAddr = gethostbyname(host) ;
if (hostAddr == NULL) {
    perror(host) ;
    return false ;
}
assert(hostAddr->h_addrtype == AF_INET) ;
memcpy(&hsrvAddr.sin_addr, hostAddr->h_addr_list[0], hostAddr->h_length) ;

int err = bind(bslServiceSock, (struct sockaddr const *)&hsrvAddr,
        sizeof(hsrvAddr)) ;
if (err == -1) {
    perror("bind()") ;
    return false ;
}
```

❶ Note we bind to `localhost`. Remote communications to a test harness is not supported.

By default we fix the port number on which the service receives connections. It is easiy enought to change on the command line if necessary.

```
<<harness constants>>=
#ifndef HARNESS_PORT
#   define HARNESS_PORT 3906    /* default port number for TCP connection */
#endif /* HARNESS_PORT */
```

In addition to listening on the socket, we register the file descriptor with the `micca` run-time. The POSIX version of the `micca` run-time will perform callbacks when file descriptors change state.

```
<<bslInitialize: listen on socket>>=
mrt_RegisterFDService(bslServiceSock, bslAcceptConnection, NULL, NULL) ;
err = listen(bslServiceSock, 1) ;
if (err == -1) {
    perror("listen()") ;
    return false ;
}
```

We also want to handle the usual signals that indicate the harness should be shut down. Again the `micca` run-time for POSIX supplies functions to interface signals to the event loop and obtain a callback when the signal occurs.

```
<<bslInitialize: register signals>>=
mrt_RegisterSignal(SIGQUIT, bslExitOnSignal) ;
mrt_RegisterSignal(SIGINT, bslExitOnSignal) ;
mrt_RegisterSignal(SIGTERM, bslExitOnSignal) ;
```

Accepting service connections

When a connection attempt occurs on the test harness socket, control is transferred by the event loop to the `bslAcceptConnection` function.

```
<<harness forward function declarations>>=
static void bslAcceptConnection(int sock) ;
```

The implementation is a bit different than is usually used for a server program. First, a test harness is run for only a single session. It does **not** service multiple clients, so the passive listening socket is closed after the client connection is accepted. Once the client connection is established, it is also registered with the `micca` run-time to receive commands.

```
<<harness static function definitions>>=
static void
bslAcceptConnection(
    int sock)
{
    int fd = accept(sock, NULL, 0) ;

    int err = shutdown(sock, SHUT_RDWR) ;
    if (err == -1) {
        perror("shutdown()") ;
        exit(EXIT_FAILURE) ;
    }
    mrt_UnregisterFDService(sock, true, false, false) ;
    bslServiceSock = -1 ;

    if (fd == -1) {
        perror("accept()") ;
        exit(EXIT_FAILURE) ;
    }

    mrt_RegisterFDService(fd, bslReadAndEvalCommand, NULL, NULL) ;
    bslCmdStream = fdopen(fd, "w+") ;                                    // ❶
    err = setvbuf(bslCmdStream, NULL, _IOLBF, 0) ;                       // ❷
}
```

❶ It is easiest just to deal with the communications connection as a standard stream.

❷ We want line buffering so that the response is sent on an ASCII record basis.

The command stream is stored in a file scoped variable so that the I/O functions can just use it directly. Only handling a single client makes all this much simpler.

```
<<harness static data>>=
static FILE *bslCmdStream ;
```

Evaluating command input

The `bslReadAndEvalCommand` function was registered as the callback for read activity on the communications connection.

```
<<harness forward function declarations>>=
static void bslReadAndEvalCommand(int rfd) ;
```

The `micca` run-time code arranges for this function to be called whenever there is data to be read on the connection. The function reads a line of input and treats it as a command, parsing the command arguments, executing the command, and formulating a response.

```
<<harness static function definitions>>=
static void
bslReadAndEvalCommand(
    int rfd)
{
```

```
    <<blsReadAndEvalCommand: read command>>
    <<blsReadAndEvalCommand: initialize response>>
    <<blsReadAndEvalCommand: parse and execute command>>
    <<blsReadAndEvalCommand: finish response>>
}
```

The command is read using `fgets`. There is some additional code to handle end of file and read errors.

```
<<blsReadAndEvalCommand: read command>>=
static char cmdBuffer[BUFSIZ] ;

char *cmd = fgets(cmdBuffer, sizeof(cmdBuffer), bslCmdStream) ;
if (cmd == NULL) {
    if (feof(bslCmdStream)) {
        dbgprintf("goodbye\n") ;
        bslClose() ;
        exit(EXIT_SUCCESS) ;
    } else if (ferror(bslCmdStream)) {
        if (errno == EINTR) {           // ❶
            clearerr(bslCmdStream) ;
            return ;
        }
        perror("fgets()") ;
        bslClose() ;
        exit(EXIT_FAILURE) ;
    } else {
        dbgprintf("partial\n") ;
        return ;
    }
}
bslTrimSpaceRight(cmdBuffer) ;          // ❷
dbgprintf("%s\n", cmdBuffer) ;
```

❶ Allow for interrupted reads on the socket.

❷ Remove any trailing whitespace.

The response to a command is built up word by word and a set of buffered contexts is used to hold a word as it it being generated. We discuss these functions in another section.

```
<<blsReadAndEvalCommand: initialize response>>=
bslBeginResponse(CmdContext) ;
bslAddWordToContext(CmdContext, "cmd") ;
bslPushContext(CmdContext) ;
```

The parsing strategy is to accumlate an array of pointers to the character strings that are the words in the command. The parsing is done, *in situ*, by writing `NUL` characters over the whitespace separators in the command and recording the addresses of where words begin.

```
<<blsReadAndEvalCommand: parse and execute command>>=
static char parseBuffer[BUFSIZ] ;
strcpy(parseBuffer, cmdBuffer) ;                        // ❶

int argc = 0 ;
char *const *argv = NULL ;
int err = bslParseCommand(parseBuffer, &argc, &argv) ;

if (err == -1) {
    bslFormatToContext(CmdContext, "name unknown status error result ") ;
```

```
    bslPushContext(CmdContext) ;
    bslFormatToContext(CmdContext,
            "failed to parse command: \"%s\"", cmdBuffer) ;
} else if (argc != 0) {
    bslExecCommand(argc, argv) ;
} else {
    return ;
}
```

❶ We copy the command before parsing it so that we can have the original string to use in case of an error.

```
<<blsReadAndEvalCommand: finish response>>=
bslEndResponse(CmdContext) ;
```

Clean up and exit

When common signals are caught, we clean up and exit.

```
<<harness forward function declarations>>=
static void bslExitOnSignal(int signal) ;
```

```
<<harness static function definitions>>=
static void
bslExitOnSignal(
    int signal)
{
    dbgprintf("caught signal %d\n", signal) ;
    bslClose() ;
    exit(EXIT_SUCCESS) ;
}
```

Closing down requires both closing the socket and unregistering the file descriptor from the `micca` run-time. This is a bit overdone, since if the intent is to terminate the process, then the operating system will take care of any loose ends.

```
<<harness forward function declarations>>=
static void bslClose(void) ;
```

```
<<harness static function definitions>>=
static void
bslClose(void)
{
    if (bslServiceSock != -1) {
        shutdown(bslServiceSock, SHUT_RDWR) ;
        bslServiceSock = -1 ;
    }
    if (bslCmdStream != NULL) {
        mrt_UnregisterFDService(fileno(bslCmdStream), true, false, false) ;
        fclose(bslCmdStream) ;
    }
}
```

Parsing harness commands

The command parser is a very simplistic state machine implemented in a `switch` statement. The parsing states determine how to handle escape characters and other special situation used to determine word boundaries.

```
<<harness forward function declarations>>=
static int bslParseCommand(char *line, int *pargc, char *const **pargv) ;
```

```
<<harness static function definitions>>=
static int
bslParseCommand(
    char *line,
    int *pargc,
    char *const **pargv)
{
    #define MAX_ARGS    64
    static char *args[MAX_ARGS] ;

    char **pArgv = args ;
    char **const argvEnd = pArgv + MAX_ARGS ;

    int quoteCount = 0 ;
    int parseState = OUTOFWORD ;
    int parseStackStorage[2] ;
    int *parseStackTop = parseStackStorage ;

    char *pWord = line ;
    for ( ; *line != ASCII_NUL ; line++) {
        char c = *line ;
        switch (parseState) {
        case OUTOFWORD:
            if (isgraph(c)) {
                if (c == ESCAPE_CHAR) {
                    *parseStackTop++ = INWORD ;
                    parseState = INESCAPE ;
                    if (pArgv < argvEnd) {
                        *pArgv++ = ++pWord ;
                    } else {
                        return -1 ;
                    }
                } else if (c == BEGIN_QUOTE) {
                    *parseStackTop++ = OUTOFWORD ;
                    parseState = INQUOTE ;
                    ++quoteCount ;
                    if (pArgv < argvEnd) {
                        *pArgv++ = ++pWord ;
                    } else {
                        return -1 ;
                    }
                } else {
                    // new word beginning
                    if (pArgv < argvEnd) {
                        *pArgv++ = pWord ;
                    } else {
                        fprintf(stderr, "exceeded max arguments") ;
                        return -1 ;
                    }
                    *pWord++ = c ;
                    parseState = INWORD ;
                }
            } // else the character is just space to skip
            break ;

        case INWORD:
            if (c == ESCAPE_CHAR) {
                *parseStackTop++ = parseState ;
```

```
            parseState = INESCAPE ;
        } else if (c == BEGIN_QUOTE) {
            *parseStackTop++ = parseState ;
            parseState = INQUOTE ;
            ++quoteCount ;
        } else if (isspace(c)) {
            // end of word
            parseState = OUTOFWORD ;
            *pWord++ = ASCII_NUL ;
        } else {
            *pWord++ = c ;
        }
        break ;

    case INESCAPE:
        switch (c) {
            case 'a':
                *pWord++ = '\a' ;
                break ;
            case 'b':
                *pWord++ = '\b' ;
                break ;
            case 't':
                *pWord++ = '\t' ;
                break ;
            case 'n':
                *pWord++ = '\n' ;
                break ;
            case 'v':
                *pWord++ = '\v' ;
                break ;
            case 'f':
                *pWord++ = '\f' ;
                break ;
            case 'r':
                *pWord++ = '\r' ;
                break ;
            default:
                *pWord++ = c ;
                break ;
        }
        parseState = *--parseStackTop ;
        break ;

    case INQUOTE:
        if (c == BEGIN_QUOTE) {
            ++quoteCount ;
            *pWord++ = c ;
        } else if (c == END_QUOTE) {
            if (--quoteCount == 0) {
                parseState = *--parseStackTop ;
                // check if ending the quote also ended the word
                if (parseState == OUTOFWORD) {
                    *pWord++ = ASCII_NUL ;
                }
            } else {
                *pWord++ = c ;
            }
        } else if (c == ESCAPE_CHAR) {
            *parseStackTop++ = parseState ;
            parseState = INESCAPE ;
        } else {
```

```
                *pWord++ = c ;
            }
            break ;
        }
    }

    *pWord = ASCII_NUL ;
    *pArgv = NULL ;
    if (quoteCount == 0 && parseState != INESCAPE) {
        if (pargc) {
            *pargc = pArgv - args ;
        }
        if (pargv) {
            *pargv = args ;
        }
        return 0 ;
    } else {
        return -1 ;
    }
}
```

Executing harness commands

To execute a harness command, the command name is mapped to a function pointer. This is accomplished by a binary search on an array of structures ordered by command name giving the corresponding command function.

```
<<harness forward function declarations>>=
static void bslExecCommand(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static void
bslExecCommand(
    int argc,
    char *const *argv)
{
    bslFormatToContext(CmdContext, "name %s result ", argv[0]) ;
    bslPushContext(CmdContext) ;

    BSL_CommandMap key = {
        .name = argv[0],
        .commandFunc = NULL
    } ;

    BSL_CommandMap const *cmd =
            bsearch(&key, commands, COUNTOF(commands), sizeof(commands[0]),
            bslCommandMapCompare) ;

    bool success ;
    if (cmd != NULL) {
        assert(cmd->commandFunc != NULL) ;
        success = cmd->commandFunc(argc, argv) ;
    } else {
        bslFormatToContext(CmdContext, "unknown command, \"%s\"", argv[0]) ;
        success = false ;
    }

    bslPopContext(CmdContext) ;
    bslFormatToContext(CmdContext, "status %s", success ? "ok" : "error") ;
}
```

The mapping from command name to command function pointer is accomplished by a simple structure containing both pieces of information.

```
<<harness data types>>=
typedef bool BSL_CommandFunc(int, char * const*) ;

typedef struct bslcommandmap {
    char const *name ;
    BSL_CommandFunc *commandFunc ;
} BSL_CommandMap ;
```

When using a binary search, we need a comparison functions. This one uses `strcmp(3)` to do the actual command name comparison.

```
<<harness forward function declarations>>=
static int bslCommandMapCompare(void const *m1, void const *m2) ;
```

```
<<harness static function definitions>>=
static int
bslCommandMapCompare(
    void const *m1,
    void const *m2)
{
    BSL_CommandMap const *cm1 = m1 ;
    BSL_CommandMap const *cm2 = m2 ;

    return strcmp(cm1->name, cm2->name) ;
}
```

Finally, the command name to command function map is held in an array variable of file static scope. The array entries must be in order of ascending alphabetic order of the command name.

```
<<harness static data>>=
static BSL_CommandMap const commands[] = { // must be in ascending "name" order
    {
        .name = "cancel",
        .commandFunc = bslCancelCommand
    }, {
        .name = "create",
        .commandFunc = bslCreateCommand
    }, {
        .name = "createasync",
        .commandFunc = bslCreateAsyncCommand
    }, {
        .name = "delaysignal",
        .commandFunc = bslDelaySignalCommand
    }, {
        .name = "delete",
        .commandFunc = bslDeleteCommand
    }, {
        .name = "domainop",
        .commandFunc = bslDomainopCommand
    }, {
        .name = "eloop",
        .commandFunc = bslEloopCommand
    }, {
        .name = "instr",
        .commandFunc = bslInstrCommand
    }, {
        .name = "null",
        .commandFunc = bslNullCommand
    }, {
```

```
        .name = "query",
        .commandFunc = bslQueryCommand
    }, {
        .name = "read",
        .commandFunc = bslReadCommand
    }, {
        .name = "remaining",
        .commandFunc = bslRemainingCommand
    }, {
        .name = "signal",
        .commandFunc = bslSignalCommand
    }, {
        .name = "trace",
        .commandFunc = bslTraceCommand
    }, {
        .name = "update",
        .commandFunc = bslUpdateCommand
    }, {
        .name = "version",
        .commandFunc = bslVersionCommand
    },
} ;
```

Harness command implementation

In this section the code for the functions which implement the various harness commands is described. The command functions follow a basic pattern. Arguments arrive as an array of strings. This is the same array created when the command record was parsed into words. The argument strings may need to be converted into internal forms such as integers. Some argument strings are used directly. Once the arguments have been checked and converted, most of the command functions use `micca` run-time procedures or use the portal provided by `micca` to accomplish their goal. Each command function formats any output into a response context and returns a boolean value to indicate if the command was executed successfully.

Many of the command functions need to look up entities such as which domain or which class in a domain, etc. These searches map an external identifier, such as a class name or an event number, to an internal pointer value. The internal pointer references data needed to operate on the entity. The functions which perform the search for model level entities are described below.

Since the command functions follow a similar pattern, we present them with little additional commentary.

Cancel command

```
<<harness forward function declarations>>=
static bool bslCancelCommand(int, char *const *) ;
```

```
<<harness static function definitions>>=
static bool
bslCancelCommand(
    int argc,
    char *const *argv)
{
    if (argc != 5) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"cancel <domain> <class> <inst> <event>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
```

```
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }
    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[3] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    char *eventName = argv[4] ;
    BSL_EventMap const *event = bslFindEvent(class, eventName) ;
    if (event == NULL) {
        return false ;
    }

    MRT_EventCode eventNumber = event->number ;
    int pcode = mrt_PortalCancelDelayedEvent(portal, classId, instId,
            eventNumber) ;
    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    }

    return true ;
}
```

Create command

```
<<harness forward function declarations>>=
static bool bslCreateCommand(int, char *const *) ;
```

```
<<harness static function definitions>>=
static bool
bslCreateCommand(
    int argc,
    char *const *argv)
{
    if (argc < 3 || argc > 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"create <domain> <class> ?<state>?\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
```

```
    }

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    MRT_StateCode initState = MRT_StateCode_IG ;
    if (argc == 4) {                                                    // ❶
        char *stateName = argv[3] ;
        BSL_StateMap const *const state = bslFindState(class, stateName) ;
        if (state == NULL) {
            return false ;
        }
        initState = state->number ;
    }

    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int pcode = mrt_PortalCreateInstance(portal, classId, initState) ;
    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    }

    bslFormatToContext(CmdContext, "%d", pcode) ;

    return true ;
}
```

❶ Providing an initial state for an instance is optional.

Create async command

```
<<harness forward function declarations>>=
static bool bslCreateAsyncCommand(int, char *const *) ;
```

```
<<harness static function definitions>>=
static bool
bslCreateAsyncCommand(
    int argc,
    char *const *argv)
{
    if (argc < 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments: %d: expected, "
            "\"createasync <domain> <class> <event> ?<arg1> <arg2> ...?\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }
```

```
    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *eventName = argv[3] ;
    BSL_EventMap const *event = bslFindEvent(class, eventName) ;
    if (event == NULL) {
        return false ;
    }

    BSL_EventParamFunc *epfunc = event->paramFunc ;
    MRT_EventCode eventNumber = event->number ;
    int pcode ;
    if (epfunc != NULL) {                                          // ❶
        MRT_EventParams params ;
        bool pstatus = epfunc(argc - 4, argv + 4, params) ;
        if (!pstatus) {
            return false ;
        }
        pcode = mrt_PortalCreateInstanceAsync(portal, classId, eventNumber,
            params, sizeof(params)) ;
    } else {
        pcode = mrt_PortalCreateInstanceAsync(portal, classId, eventNumber,
            NULL, 0) ;
    }

    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    }

    bslFormatToContext(CmdContext, "%d", pcode) ;
    return true ;
}
```

❶ Parameters to the creation event are optional. Those creation events that require parameters have a generated harness function to perform the conversion from strings to internal form.

Delay signal command

```
<<harness forward function declarations>>=
static bool bslDelaySignalCommand(int, char *const *) ;
```

```
<<harness static function definitions>>=
static bool
bslDelaySignalCommand(
    int argc,
    char *const *argv)
{
    if (argc < 6) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments: %d: expected, "
            "\"delaysignal <domain> <class> <inst> <delay> <event> "
```

```
            "?<arg1> <arg2> ...?\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }
    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[3] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    char *delayTime = argv[4] ;
    MRT_DelayTime delay ;
    int converted = sscanf(delayTime, "%" SCNu32, &delay) ;
    if (converted != 1) {
        bslFormatToContext(CmdContext, failedConvertMsg, delayTime, SCNi32) ;
        return false ;
    }

    char *eventName = argv[5] ;
    BSL_EventMap const *event = bslFindEvent(class, eventName) ;
    if (event == NULL) {
        return false ;
    }

    BSL_EventParamFunc *epfunc = event->paramFunc ;
    MRT_EventCode eventNumber = event->number ;
    int pcode ;
    if (epfunc != NULL) {
        MRT_EventParams params ;
        bool pstatus = epfunc(argc - 6, argv + 6, params) ;
        if (!pstatus) {
            return false ;
        }
        pcode = mrt_PortalSignalDelayedEvent(portal, classId, instId,
                eventNumber, params, sizeof(params), delay) ;
    } else {
        pcode = mrt_PortalSignalDelayedEvent(portal, classId, instId,
                eventNumber, NULL, 0, delay) ;
    }

    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    }

    return true ;
```

```
}
```

Delete command

```
<<harness forward function declarations>>=
static bool bslDeleteCommand(int, char *const *) ;
```

```
<<harness static function definitions>>=
static bool
bslDeleteCommand(
    int argc,
    char *const *argv)
{
    if (argc < 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"delete <domain> <class> <inst>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[3] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int pcode = mrt_PortalDeleteInstance(portal, classId, instId) ;
    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    }

    return true ;
}
```

Domain operation command

```
<<harness forward function declarations>>=
static bool bslDomainopCommand(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslDomainopCommand(
    int argc,
    char *const *argv)
{
    if (argc < 3) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments: %d: expected, "
            "domainop <domain> <operation> ?<arg1> <arg2> ...?",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *opName = argv[2] ;
    BSL_OperationMap const *const opMap = bslFindOperation(harness, opName) ;
    if (opMap == NULL) {
        return false ;
    }
    return opMap->func(argc - 3, argv + 3) ;                // ❶
}
```

❶ The invocation of domain operations is wrapped by a `bosal` generated function which performs the conversion of the operation arguments from strings to internal form.

Event loop Command

The event loop command function is more complicated than others. The command packs substantial semantics into the argument combinations and the event loop actions must be separated out.

```
<<harness forward function declarations>>=
static bool bslEloopCommand(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslEloopCommand(
    int argc,
    char *const *argv)
{
    static char const runMsg[] = "event loop is running" ;

    bool status = true ;

    if (argc == 1) {
        switch (bslHarnessState) {
        case bslHalted :
            bslAddWordToContext(CmdContext, "halted") ;
            break ;

        case bslRunning:
            bslAddWordToContext(CmdContext, "running") ;
            break ;
```

```
        default:
            bslFormatToContext(CmdContext, "unknown harness state, \"%d\"",
                    bslHarnessState) ;
            status = false ;
            break ;
        }
    } else if (argc == 2) {
        if (strcmp(argv[1], "run") == 0) {
            bslHarnessState = bslRunning ;
            bslAddWordToContext(CmdContext, "running") ;
        } else if (strcmp(argv[1], "halt") == 0) {
            mrt_SyncToEventLoop() ;
            bslHarnessState = bslHalted ;
            bslAddWordToContext(CmdContext, "halted") ;
        } else if (strcmp(argv[1], "once") == 0) {
            if (bslHarnessState != bslRunning) {
                bool didEvent = mrt_DispatchSingleEvent() ;
                bslAddWordToContext(CmdContext, didEvent ? "true" : "false") ;
            } else {
                bslAddWordToContext(CmdContext, runMsg) ;
                status = false ;
            }
        } else if (strcmp(argv[1], "toc") == 0) {
            if (bslHarnessState != bslRunning) {
                bool didEvent = mrt_DispatchThreadOfControl(true) ;
                bslAddWordToContext(CmdContext, didEvent ? "true" : "false") ;
            } else {
                bslAddWordToContext(CmdContext, runMsg) ;
                status = false ;
            }
        } else {
            bslFormatToContext(CmdContext,
                "unknown eloop option, \"%s\": expected, "
                "\"run | halt | once | toc\"",
                argv[1]) ;
            status = false ;
        }
    } else if (argc == 3) {
        if (strcmp(argv[1], "toc") == 0) {
            if (bslHarnessState == bslRunning) {
                bslAddWordToContext(CmdContext, runMsg) ;
                status = false ;
            } else {
                if (strcmp(argv[2], "wait") == 0) {
                    bool didEvent = mrt_DispatchThreadOfControl(true) ;
                    bslAddWordToContext(CmdContext, didEvent ? "true" : "false") ;
                } else if (strcmp(argv[2], "nowait") == 0) {
                    bool didEvent = mrt_DispatchThreadOfControl(false) ;
                    bslAddWordToContext(CmdContext, didEvent ? "true" : "false") ;
                } else {
                    bslFormatToContext(CmdContext,
                        "unknown toc option, \"%s\": expected, "
                        "\"toc wait | nowait\"", argv[2]) ;
                    status = false ;
                }
            }
        } else {
            bslFormatToContext(CmdContext,
                "bad eloop command format, \"%s\": expected, "
                "\"toc ?wait | nowait?\"",
                argv[1]) ;
            status = false ;
```

```
        }
    } else {
        bslFormatToContext(CmdContext,
            "wrong # of arguments: %d: expected, "
            "\"eloop ?run | halt | once | toc ?wait | nowait?\"",
            argc) ;
        status = false ;
    }

    return status ;
}
```

Instrument command

```
<<harness forward function declarations>>=
static bool bslInstrCommand(int argc, char *const *argv) ;
```

We want to be able to control whether instrumentation output is generated. This makes it easier for simple clients to interact with a test harness without having to account of asynchronous output from the harness. We keep the state in a simple boolean variable.

```
<<harness static data>>=
static bool instrOn = false ;
```

```
<<harness static function definitions>>=
static bool
bslInstrCommand(
    int argc,
    char *const *argv)
{
    bool status = true ;

    if (argc == 1) {
        bslAddWordToContext(CmdContext, instrOn ? "on" : "off") ;
    } else if (argc == 2) {
        if (strcmp(argv[1], "on") == 0) {
            instrOn = true ;
            bslAddWordToContext(CmdContext, "on") ;
        } else if (strcmp(argv[1], "off") == 0) {
            instrOn = false ;
            bslAddWordToContext(CmdContext, "off") ;
        } else {
            bslFormatToContext(CmdContext,
                    "unknown instr option, \"%s\": expected, \"on | off\"",
                    argv[1]) ;
            status = false ;
        }
    } else {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, \"instr ?on | off?\"", argc) ;
        status = false ;
    }

    return status ;
}
```

Null command

```
<<harness forward function declarations>>=
static bool bslNullCommand(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslNullCommand(
    int argc,
    char *const *argv)
{
    if (argc != 1) {
        bslFormatToContext(CmdContext,
                "wrong # of arguments, %d: expected, null", argc) ;
        return false ;
    }

    return true ;
}
```

Query command

```
<<harness forward function declarations>>=
static bool bslQueryCommand(int, char *const *) ;
```

The query command allows queries for a number of different entities in the test harness. The implementation delegates the queries for the different entities to different functions. The data structure below is used to map the query subcommand to the function that executes the query.

```
<<harness static data>>=
static struct qsubMapEntry {
    char const *subCmdName ;
    bool (*subCmdFunc)(int, char *const *) ;
} const querySubCmdMap[] = {
    {"attributes", bslQueryAttrs},              // ❶
    {"classes", bslQueryClasses},
    {"current", bslQueryCurrent},
    {"domains", bslQueryDomains},
    {"doparams", bslQueryDomainOpParams},
    {"events", bslQueryEvents},
    {"evparams", bslQueryEventParams},
    {"instances", bslQueryInsts},
    {"operations", bslQueryDomainOps},
    {"states", bslQueryStates},
} ;
```

❶ Must be kept in alphabetical order! `bsearch()` is used to locate the query function.

Since we intend to use `bsearch` to locate the sub command function, we must then supply a comparison function for the mapping entries.

```
<<harness forward function declarations>>=
static int subMapEntryCompare(void const *m1, void const *m2) ;
```

```
<<harness static function definitions>>=
static int
subMapEntryCompare(
    void const *m1,
    void const *m2)
```

```
{
    struct qsubMapEntry const *sm1 = m1 ;
    struct qsubMapEntry const *sm2 = m2 ;

    return strcmp(sm1->subCmdName, sm2->subCmdName) ;
}
```

```
<<harness static function definitions>>=
static bool
bslQueryCommand(
    int argc,
    char *const *argv)
{
    if (argc < 2) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "query <subcmd> ?<domain> <arg1> <arg2> ...?",
            argc) ;
        return false ;
    }

    struct qsubMapEntry key = {
        .subCmdName = argv[1],
        .subCmdFunc = NULL
    } ;

    struct qsubMapEntry const *const subMapEntry =
            bsearch(&key, querySubCmdMap, COUNTOF(querySubCmdMap),
            sizeof(key), subMapEntryCompare) ;
    if (subMapEntry == NULL) {
        bslFormatToContext(CmdContext,
            "unknown subcommand, \"%s\": expected, "
            "\"domains | classes | operations | doparams | attributes |"
            " instances | states | events | evparams | current\"", argv[1]) ;
        return false ;
    }

    assert(subMapEntry->subCmdFunc != NULL) ;
    return subMapEntry->subCmdFunc(argc, argv) ;
}
```

Query domains

```
<<harness forward function declarations>>=
static bool bslQueryDomains(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryDomains(
    int argc,
    char *const *argv)
{
    if (argc != 2) {
        bslFormatToContext(CmdContext,
                "wrong # of arguments, %d: expected, \"query domains\"", argc) ;
        return false ;
    }

    BSL_DomainHarness const *harness = bslHarnesses ;
```

```
    for (int count = bslHarnessCount ; count > 0 ; count--) {
        assert(harness->portal != NULL) ;
        bslAddWordToContext(CmdContext, mrt_PortalDomainName(harness->portal)) ;
        harness++ ;
    }
    return true ;
}
```

Query domain operations

```
<<harness forward function declarations>>=
static bool bslQueryDomainOps(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryDomainOps(
    int argc,
    char *const *argv)
{
    if (argc != 3) {
        bslFormatToContext(CmdContext,
                "wrong # of arguments, %d: expected, "
                "\"query operations <domain>\"", argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    bslPushContext(CmdContext) ;
    BSL_OperationMap const *opMap = harness->operations ;
    for (int cnt = harness->operationCount ; cnt > 0 ; cnt--) {
        bslAddWordToContext(CmdContext, opMap->name) ;

        opMap++ ;
    }
    bslPopContext(CmdContext) ;

    return true ;
}
```

Query domain operation parameters

```
<<harness forward function declarations>>=
static bool bslQueryDomainOpParams(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryDomainOpParams(
    int argc,
    char *const *argv)
{
    if (argc != 4) {
        bslFormatToContext(CmdContext,
                "wrong # of arguments, %d: expected, "
```

```
                "\"query doparams <domain> <operation>\"", argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *opName = argv[3] ;
    BSL_OperationMap const *const opMap = bslFindOperation(harness, opName) ;
    if (opMap == NULL) {
        return false ;
    }

    bslPushContext(CmdContext) ;
    BSL_DomainOpParam const *params = opMap->params ;
    for (int cnt = opMap->paramCount ; cnt > 0 ; cnt--) {
        bslAddWordToContext(CmdContext, params->name) ;
        bslAddWordToContext(CmdContext, params->dataType) ;

        params++ ;
    }
    bslPopContext(CmdContext) ;

    return true ;
}
```

Query classes

```
<<harness forward function declarations>>=
static bool bslQueryClasses(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryClasses(
    int argc,
    char *const *argv)
{
    if (argc != 3) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, \"query classes <domain>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }
    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int classCount = mrt_PortalDomainClassCount(portal) ;
    for (MRT_ClassId id = 0 ; id < classCount ; id++) {                    // ❶
        char const *className = NULL ;
        int pcode = mrt_PortalClassName(portal, id, &className) ;
        if (pcode == 0) {
```

```
            assert(className != NULL) ;
            bslAddWordToContext(CmdContext, className) ;
        } else {
            bslResetContext(CmdContext) ;
            bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
            return false ;
        }
    }
    return true ;
}
```

❶ Class numbers always run from 0 to `classCount - 1`. The `micca` code generation for the domain guarantees that. In truth, the class id's are just array index values.

Query attributes

```
<<harness forward function declarations>>=
static bool bslQueryAttrs(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryAttrs(
    int argc,
    char *const *argv)
{
    if (argc != 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"query attributes <domain> <class>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *className = argv[3] ;
    BSL_ClassMap const *const classMap = bslFindClass(harness, className) ;
    if (classMap == NULL) {
        return false ;
    }
    MRT_ClassId classId = classMap->id ;


    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int attrCount = mrt_PortalClassAttributeCount(portal, classId) ;
    assert(attrCount >= 0) ;
    if (attrCount < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(attrCount)) ;
        return false ;
    }

    for (MRT_AttrId attrId = 0 ; attrId < attrCount ; attrId++) {
        char const *attrName = NULL ;
```

```
        int pcode = mrt_PortalClassAttributeName(portal, classId, attrId,
                &attrName) ;
        if (pcode == 0) {
            assert(attrName != NULL) ;
            bslAddWordToContext(CmdContext, attrName) ;
        } else {
            bslResetContext(CmdContext) ;
            bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
            return false ;
        }
    }

    return true ;
}
```

Query instances

```
<<harness forward function declarations>>=
static bool bslQueryInsts(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryInsts(
    int argc,
    char *const *argv)
{
    if (argc != 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"query instances <domain> <class>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *className = argv[3] ;
    BSL_ClassMap const *const classMap = bslFindClass(harness, className) ;
    if (classMap == NULL) {
        return false ;
    }
    MRT_ClassId classId = classMap->id ;


    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int instCount = mrt_PortalClassInstanceCount(portal, classId) ;
    assert(instCount >= 0) ;
    if (instCount < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(instCount)) ;
        return false ;
    }

    bslFormatToContext(CmdContext, "total %d named ", instCount) ;
```

```
    bslPushContext(CmdContext) ;
    BSL_InstMap const *inst = classMap->insts ;
    for (unsigned namedCount = classMap->instCount ; namedCount != 0 ;
            namedCount--, inst++) {
        bslAddWordToContext(CmdContext, inst->name) ;
        bslFormatToContext(CmdContext, "%u ", inst->id) ;
    }
    bslPopContext(CmdContext) ;

    return true ;
}
```

Query states

```
<<harness forward function declarations>>=
static bool bslQueryStates(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryStates(
    int argc,
    char *const *argv)
{
    if (argc != 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"query states <domain> <class>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *className = argv[3] ;
    BSL_ClassMap const *const classMap = bslFindClass(harness, className) ;
    if (classMap == NULL) {
        return false ;
    }
    MRT_ClassId classId = classMap->id ;


    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int stateCount = mrt_PortalClassStateCount(portal, classId) ;
    if (stateCount > 0) {
        for (MRT_StateCode stateId = 0 ; stateId < stateCount ; stateId++) {
            char const *stateName = NULL ;
            int pcode = mrt_PortalClassStateName(portal, classId, stateId,
                    &stateName) ;
            if (pcode == 0) {
                assert(stateName != NULL) ;
                bslAddWordToContext(CmdContext, stateName) ;
            } else {
                bslResetContext(CmdContext) ;
                bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
```

```
                return false ;
            }
        }
    }

    return true ;
}
```

Query events

```
<<harness forward function declarations>>=
static bool bslQueryEvents(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryEvents(
    int argc,
    char *const *argv)
{
    if (argc != 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"query events <domain> <class>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *className = argv[3] ;
    BSL_ClassMap const *const classMap = bslFindClass(harness, className) ;
    if (classMap == NULL) {
        return false ;
    }
    MRT_ClassId classId = classMap->id ;


    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int eventCount = mrt_PortalClassEventCount(portal, classId) ;
    if (eventCount > 0) {
        for (MRT_EventCode eventId = 0 ; eventId < eventCount ; eventId++) {
            char const *eventName = NULL ;
            int pcode = mrt_PortalClassEventName(portal, classId, eventId,
                    &eventName) ;
            if (pcode == 0) {
                assert(eventName != NULL) ;
                bslAddWordToContext(CmdContext, eventName) ;
            } else {
                bslResetContext(CmdContext) ;
                bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
                return false ;
            }
        }
    }
```

```
    return true ;
}
```

Query event parameters

```
<<harness forward function declarations>>=
static bool bslQueryEventParams(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryEventParams(
    int argc,
    char *const *argv)
{
    if (argc != 5) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"query evparams <domain> <class> <event>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *className = argv[3] ;
    BSL_ClassMap const *const classMap = bslFindClass(harness, className) ;
    if (classMap == NULL) {
        return false ;
    }

    char *eventName = argv[4] ;
    BSL_EventMap const *const eventMap = bslFindEvent(classMap, eventName) ;
    if (eventMap == NULL) {
        return false ;
    }

    bslPushContext(CmdContext) ;
    BSL_EventParam const *params = eventMap->params ;
    for (int i = eventMap->paramCount ; i > 0 ; i--) {
        bslAddWordToContext(CmdContext, params->name) ;
        bslAddWordToContext(CmdContext, params->dataType) ;

        params++ ;
    }
    bslPopContext(CmdContext) ;

    return true ;
}
```

Query current state

```
<<harness forward function declarations>>=
static bool bslQueryCurrent(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslQueryCurrent(
    int argc,
    char *const *argv)
{
    if (argc != 5) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"query current <domain> <class> <instance>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[2] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    char *className = argv[3] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[4] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    int statecode = mrt_PortalInstanceCurrentState(portal, classId, instId) ;
    if (statecode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(statecode)) ;
        return false ;
    }

    char const *stateName ;
    int pcode = mrt_PortalClassStateName(portal, classId, statecode, &stateName) ;
    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    }

    bslAddWordToContext(CmdContext, stateName) ;
    return true ;
}
```

Read Command

```
<<harness forward function declarations>>=
static bool bslReadCommand(int, char *const *) ;
```

The read command implementation is slightly more complicated. The command allows one to read all the attributes without having to specify their names or to read just a specified set of attributes. This causes the two cases to be separated.

```
<<harness static function definitions>>=
static bool
bslReadCommand(
    int argc,
    char *const *argv)
{
    if (argc < 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"read <domain> <class> <inst> ?<attr1> <attr2> ...?\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }

    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[3] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    bool status = true ;
    if (argc == 4) {
        // read all the attributes
        BSL_AttrMap const *attr = class->attrs ;
        for (unsigned attrCount = class->attrCount ; attrCount != 0 ;
                attrCount--, attr++) {
            bslAddWordToContext(CmdContext, attr->name) ;
            bslPushContext(CmdContext) ;
            status = attr->readFunc(portal, classId, instId, attr->id) ;
            bslPopContext(CmdContext) ;

            if (!status) {
                break ;
            }
        }
    } else {
        char *const *attrNames = argv + 4 ;
        for (int attrCount = argc - 4 ; attrCount != 0 ;
                attrCount--, attrNames++) {
            BSL_AttrMap const *const attr = bslFindAttr(class, *attrNames) ;
            if (attr == NULL) {
                bslResetContext(CmdContext) ;
                bslFormatToContext(CmdContext,
```

```
                        "unknown attribute, \"%s\"", *attrNames) ;
                status = false ;
                break ;
            }

            bslAddWordToContext(CmdContext, *attrNames) ;
            bslPushContext(CmdContext) ;
            status = attr->readFunc(portal, classId, instId, attr->id) ;
            bslPopContext(CmdContext) ;

            if (!status) {
                break ;
            }
        }
    }

    return status ;
}
```

Remaining Command

```
<<harness forward function declarations>>=
static bool bslRemainingCommand(int, char *const *) ;
```

```
<<harness static function definitions>>=
static bool
bslRemainingCommand(
    int argc,
    char *const *argv)
{
    if (argc != 5) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"remaining <domain> <class> <inst> <event>\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }
    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[3] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    char *eventName = argv[4] ;
```

```
    BSL_EventMap const *event = bslFindEvent(class, eventName) ;
    if (event == NULL) {
        return false ;
    }
    MRT_EventCode eventNumber = event->number ;

    MRT_DelayTime delay ;
    int pcode = mrt_PortalRemainingDelayTime(portal, classId, instId,
            eventNumber, &delay) ;
    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    } else {
        bslFormatToContext(CmdContext, "%" PRIu32, delay) ;
    }

    return true ;
}
```

Signal Command

```
<<harness forward function declarations>>=
static bool bslSignalCommand(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslSignalCommand(
    int argc,
    char *const *argv)
{
    if (argc < 5) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments: %d: expected, "
            "\"signal <domain> <class> <inst> <event> ?<arg1> <arg2> ...?\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
    }
    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[3] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    char *eventName = argv[4] ;
```

```
    BSL_EventMap const *event = bslFindEvent(class, eventName) ;
    if (event == NULL) {
        return false ;
    }

    BSL_EventParamFunc *epfunc = event->paramFunc ;
    MRT_EventCode eventNumber = event->number ;
    int pcode ;
    if (epfunc != NULL) {
        MRT_EventParams params ;
        bool pstatus = epfunc(argc - 5, argv + 5, params) ;
        if (!pstatus) {
            return false ;
        }
        pcode = mrt_PortalSignalEvent(portal, classId, instId, eventNumber,
            params, sizeof(params)) ;
    } else if (argc > 5) {
        bslFormatToContext(CmdContext, wrongNumArgsMsg, argc - 5, 0) ;
        return false ;
    } else {
        pcode = mrt_PortalSignalEvent(portal, classId, instId, eventNumber,
            NULL, 0) ;
    }

    if (pcode < 0) {
        bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;
        return false ;
    }

    return true ;
}
```

Trace Command

```
<<harness forward function declarations>>=
static bool bslTraceCommand(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslTraceCommand(
    int argc,
    char *const *argv)
{
    bool status = true ;

    if (argc == 1) {
        MRT_TraceHandler handler = mrt_RegisterTraceHandler(NULL) ;
        mrt_RegisterTraceHandler(handler) ;
        bslAddWordToContext(CmdContext, handler == NULL ?  "off" : "on") ;
    } else if (argc == 2) {
        if (strcmp(argv[1], "on") == 0) {
            mrt_RegisterTraceHandler(bslTraceResponse) ;
            bslAddWordToContext(CmdContext, "on") ;
        } else if (strcmp(argv[1], "off") == 0) {
            mrt_RegisterTraceHandler(NULL) ;
            bslAddWordToContext(CmdContext, "off") ;
        } else {
            bslFormatToContext(CmdContext,
                    "unknown trace option, \"%s\": expected, \"on | off\"",
                    argv[1]) ;
```

```
            status = false ;
        }
    } else {
        bslFormatToContext(CmdContext,
            "wrong # of arguments: %d: expected, \"trace ?on | off?\"", argc) ;
        status = false ;
    }

    return status ;
}
```

Responding to traces

If tracing is turned on, then the `bslTraceResponse()` function is registered with the `micca` run-time to handle output of tracing information. The code is here straight forward enough, but rather longer since it must handle the three different types of traces. The code for the three types is placed in separate literate chunks.

```
<<harness forward function declarations>>=
static void bslTraceResponse(MRT_TraceInfo const *traceInfo) ;
```

```
<<harness static function definitions>>=
static void
bslTraceResponse(
    MRT_TraceInfo const *traceInfo)
{
    <<bslTraceResponse: resolve source instance name>>

    <<bslTraceResponse: resolve target instance name>>

    <<bslTraceResponse: resolve event name>>

    bslBeginResponse(TraceContext) ;
    bslAddWordToContext(TraceContext, "trace") ;
    bslPushContext(TraceContext) ;

    <<bslTraceResponse: common trace info>>

    switch (traceInfo->eventType) {
    case mrtTransitionEvent: {
        <<bslTraceResponse: transition traces>>
    }
        break ;

    case mrtPolymorphicEvent: {
        <<bslTraceResponse: polymorphic traces>>
    }
        break ;

    case mrtCreationEvent: {
        <<bslTraceResponse: creation traces>>
    }
        break ;

    default:
        bslFormatToContext(TraceContext, "type unknown") ;
        break ;
    }

    bslEndResponse(TraceContext) ;
}
```

It is possible for an event to arise from outside a domain. In that case, we don't know the instance name or class name.

Note that we insist there there be naming information compiled into the domain and we make several assertions to effect in the code. Also note that for instances, only instance in the initial instance population have names. All others are only known by a number.

```
<<bslTraceResponse: resolve source instance name>>=
char const *sourceClassName ;
char sourceIdNum[32] ;
char const *sourceName ;

if (traceInfo->sourceInst == NULL) {
    sourceName = "?" ;
    sourceClassName = "?" ;
} else {
    sourceClassName = traceInfo->sourceInst->classDesc->name ;
    assert(sourceClassName != NULL) ;

    sourceName = traceInfo->sourceInst->name ;
    if (sourceName == NULL) {
        unsigned instid = mrt_InstanceIndex(traceInfo->sourceInst) ;
        snprintf(sourceIdNum, sizeof(sourceIdNum), "%u", instid) ;
        sourceName = sourceIdNum ;
    }
}
```

For the target instance, we also have to handle the possiblity of there being no naming information for the instance and fall back to numbers if necessary.

```
<<bslTraceResponse: resolve target instance name>>=
char targetIdNum[32] ;
char const *targetName = traceInfo->targetInst->name ;
if (targetName == NULL) {
    unsigned instid = mrt_InstanceIndex(traceInfo->targetInst) ;
    snprintf(targetIdNum, sizeof(targetIdNum), "%u", instid) ;
    targetName = targetIdNum ;
}
```

There should always be an event name since we are insisting on naming information being compiled in to the domain, and we assert that here.

```
<<bslTraceResponse: resolve event name>>=
char const *eventName ;

assert(traceInfo->targetInst->classDesc->eventNames != NULL);
eventName =
        traceInfo->targetInst->classDesc->eventNames[traceInfo->eventNumber] ;
```

All types of traces have some common information associated with them. We add that to the response first.

```
<<bslTraceResponse: common trace info>>=
bslFormatToContext(TraceContext,
    "time %s "
    "event %s "
    "source %s.%s "
    "target %s.%s ",

    bslTimestamp(),
    eventName,
    sourceClassName, sourceName,
    traceInfo->targetInst->classDesc->name, targetName) ;
```

Ordinary transition traces are the most common. They record the transition of a state machine from its current state to a new state. We have some special handling when the result of the transition is *ignored* or *can't happen*.

```
<<bslTraceResponse: transition traces>>=
MRT_StateCode newState = traceInfo->info.transitionTrace.newState ;
char const *newStateName ;
if (newState == MRT_StateCode_IG) {
    newStateName = "IG" ;
} else if (newState == MRT_StateCode_CH) {
    newStateName = "CH" ;
} else {
    newStateName = traceInfo->targetInst->classDesc->edb->stateNames[
        traceInfo->info.transitionTrace.newState] ;
}

bslFormatToContext(TraceContext,
    "type transition "
    "currstate %s "
    "newstate %s ",

    traceInfo->targetInst->classDesc->edb->stateNames[
        traceInfo->info.transitionTrace.currentState],
    newStateName) ;
```

Polymorphic event traces contain information about which generalization relationship is traversed and the new subclass type to which the event is directed. Polymorphic event traces are followed by zero or more additional polymorphic event traces and finally be an ordinary transition trace.

```
<<bslTraceResponse: polymorphic traces>>=
MRT_Relationship const *rel = traceInfo->targetInst->classDesc->pdb->
        genDispatch[traceInfo->info.polyTrace.genNumber].relship ;
MRT_Class const *subclass ;
char const *subname = NULL ;
if (rel->relType == mrtRefGeneralization) {
    subclass = rel->relInfo.refGeneralization.
        subclasses[traceInfo->info.polyTrace.subcode].classDesc ;
    subname = subclass->name ;
} else if (rel->relType == mrtUnionGeneralization) {
    subclass = rel->relInfo.unionGeneralization.
        subclasses[traceInfo->info.polyTrace.subcode] ;
    subname = subclass->name ;
} else {
    subname = "unknown" ;
    break ;
}

bslFormatToContext(TraceContext,
    "type polymorphic "
    "relationship %s "
    "newevent %s "
    "subclass %s ",

    traceInfo->targetInst->classDesc->pdb->genNames[
        traceInfo->info.polyTrace.genNumber],
    subclass->eventNames[traceInfo->info.polyTrace.mappedEvent],
    subname) ;
```

Creation traces have no additional information. The target of the event is the newly created instance. The creation trace is followed by a transition trace as the newly created instance transitions from its pseudo-initial state.

```
<<bslTraceResponse: creation traces>>=
bslFormatToContext(TraceContext, "type creation") ;
```

Traces are timestamped before sending to the test harness client. The standard library time functions handle the heavy lifting.

```
<<harness forward function declarations>>=
static char const *bslTimestamp(void) ;
```

```
<<harness static function definitions>>=
static char const *
bslTimestamp(void)
{
    static char timestamp[128] ;

    struct timeval now ;
    if (gettimeofday(&now, NULL) != 0) {
        return "unknown" ;
    }

    struct tm *ltime ;
    ltime = localtime(&now.tv_sec) ;
    if (ltime == NULL) {
        return strerror(errno) ;
    }

    int tlen = strftime(timestamp, sizeof(timestamp), "%FT%T", ltime) ;
    if (tlen == 0) {
        return strerror(errno) ;
    }

    int flen = snprintf(timestamp + tlen, sizeof(timestamp) - tlen,
            ".%03u.%03u", (unsigned)(now.tv_usec / 1000),
            (unsigned)(now.tv_usec % 1000)) ;
    if (flen > (sizeof(timestamp) - tlen)) {
        return "timestamp too big" ;
    }

    return timestamp ;
}
```

Update Command

```
<<harness forward function declarations>>=
static bool bslUpdateCommand(int, char *const *) ;
```

```
<<harness static function definitions>>=
static bool
bslUpdateCommand(
    int argc,
    char *const *argv)
{
    if (argc < 4) {
        bslFormatToContext(CmdContext,
            "wrong # of arguments, %d: expected, "
            "\"update <domain> <class> <inst> ?<attr1> <value1> ...?\"",
            argc) ;
        return false ;
    }

    char *domainName = argv[1] ;
    BSL_DomainHarness const *const harness = bslFindHarness(domainName) ;
    if (harness == NULL) {
        return false ;
```

```
    }

    MRT_DomainPortal const *portal = harness->portal ;
    assert(portal != NULL) ;

    char *className = argv[2] ;
    BSL_ClassMap const *const class = bslFindClass(harness, className) ;
    if (class == NULL) {
        return false ;
    }
    MRT_ClassId classId = class->id ;

    char *instName = argv[3] ;
    int instId = bslFindInst(class, instName) ;
    if (instId < 0) {
        return false ;
    }

    bool status = true ;
    int updateCount = argc - 4 ;
    if (updateCount % 2 != 0) {
        bslFormatToContext(CmdContext,
                "updates must be given as name / value pairs") ;
        return false ;
    }

    char *const *attrName = argv + 4 ;
    char *const *valueString = argv + 5 ;
    for ( ; updateCount != 0 ;
            updateCount -= 2, attrName += 2, valueString += 2) {
        BSL_AttrMap const *const attr = bslFindAttr(class, *attrName) ;
        if (attr == NULL) {
            bslResetContext(CmdContext) ;
            bslFormatToContext(CmdContext,
                    "unknown attribute, \"%s\"", *attrName) ;
            status = false ;
            break ;
        }

        bslAddWordToContext(CmdContext, *attrName) ;
        bslPushContext(CmdContext) ;
        status = attr->updateFunc(portal, classId, instId, attr->id,
                *valueString) ;
        if (status) {
            status = attr->readFunc(portal, classId, instId, attr->id) ;
            bslPopContext(CmdContext) ;
        }

        if (!status) {
            break ;
        }
    }

    return status ;
}
```

Version command

```
<<harness forward function declarations>>=
static bool bslVersionCommand(int argc, char *const *argv) ;
```

```
<<harness static function definitions>>=
static bool
bslVersionCommand(
    int argc,
    char *const *argv)
{
    if (argc != 1) {
        bslFormatToContext(CmdContext, wrongNumArgsMsg, 0, 1) ;
        return false ;
    }

    bslAddWordToContext(CmdContext, BOSAL_VERSION) ;
    return true ;
}
```

Locating Domain Components

Much of the function of a test harness is to map external identifiers onto internal data. For example, domains are known by a name, which is a string, outside of the test harness, but operations on the domain require a pointer to the portal for the domain.

In this section we describe the various functions used to perform the mapping of strings and numbers into pointers to the internal data structures used in the actual operations on the domain components.

Note also that if these search functions fail, they place an error message into the command response context indicating the cause of the failure. This prevents having to generate error messages at invocation of the search functions.

Finding a Harness

The harness code generator creates a variable which is an array of data structures containing a description of the domains integrated into the harness. This array is not created in any particular order, so finding a particular domain in the test harness is done using a simple linear sequential search. Since the number of domain in a test harness is usually quite small (usually much less than even 10), the search method is not critical to the performance.

```
<<harness forward function declarations>>=
static BSL_DomainHarness const *const bslFindHarness(char const *domainName) ;
```

```
<<harness static function definitions>>=
static BSL_DomainHarness const *const
bslFindHarness(
    char const *domainName)
{
    assert(domainName != NULL) ;

    BSL_DomainHarness const *hiter = bslHarnesses ;
    int hcount = bslHarnessCount ;
    for ( ; hcount != 0 ; hcount--, hiter++) {
        assert(hiter->portal != NULL) ;
        assert(hiter->portal->name != NULL) ;
        char const *trialName = mrt_PortalDomainName(hiter->portal) ;
        if (strcmp(domainName, trialName) == 0) {
            return hiter ;
        }
    }
    bslFormatToContext(CmdContext, "unknown domain, \"%s\"", domainName) ;
    return NULL ;
}
```

Finding a Domain Operation

The mapping data of domain operation name to its corresponding operation function is ordered by ascending operation name, so a binary search may be used to find a particular operation name.

```
<<harness forward function declarations>>=
static BSL_OperationMap const *const bslFindOperation(
    BSL_DomainHarness const *const harness, char const *opName) ;
```

```
<<harness static function definitions>>=
static BSL_OperationMap const *const
bslFindOperation(
    BSL_DomainHarness const *const harness,
    char const *opName)
{
    BSL_OperationMap key = {
        .name = opName,
        .func = NULL
    } ;

    BSL_OperationMap const *const opMap =
            bsearch(&key, harness->operations, harness->operationCount,
            sizeof(key), bslOperationMapCompare) ;
    if (opMap == NULL) {
        bslFormatToContext(CmdContext, "unknown operation, \"%s\"", opName) ;
    }

    return opMap ;
}
```

```
<<harness forward function declarations>>=
static int bslOperationMapCompare(void const *m1, void const *m2) ;
```

```
<<harness static function definitions>>=
static int
bslOperationMapCompare(
    void const *m1,
    void const *m2)
{
    BSL_OperationMap const *cm1 = m1 ;
    BSL_OperationMap const *cm2 = m2 ;

    return strcmp(cm1->name, cm2->name) ;
}
```

Finding a Domain Class

```
<<harness forward function declarations>>=
static BSL_ClassMap const *const bslFindClass(
    BSL_DomainHarness const *const harness, char const *className) ;
```

```
<<harness static function definitions>>=
static BSL_ClassMap const *const
bslFindClass(
    BSL_DomainHarness const *const harness,
    char const *className)
{
    BSL_ClassMap key = {
```

```
        .name = className
    } ;

    BSL_ClassMap const *const classMap =
            bsearch(&key, harness->classes, harness->classCount,
            sizeof(key), bslClassMapCompare) ;
    if (classMap == NULL) {
        bslFormatToContext(CmdContext, "unknown class, \"%s\"", className) ;
    }

    return classMap ;
}
```

```
<<harness forward function declarations>>=
static int bslClassMapCompare(void const *m1, void const *m2) ;
```

```
<<harness static function definitions>>=
static int
bslClassMapCompare(
    void const *m1,
    void const *m2)
{
    BSL_OperationMap const *cm1 = m1 ;
    BSL_OperationMap const *cm2 = m2 ;

    return strcmp(cm1->name, cm2->name) ;
}
```

Finding a Class Attribute

```
<<harness forward function declarations>>=
static BSL_AttrMap const *const
bslFindAttr(
    BSL_ClassMap const *const class,
    char const *attrName) ;
```

```
<<harness static function definitions>>=
static BSL_AttrMap const *const
bslFindAttr(
    BSL_ClassMap const *const class,
    char const *attrName)
{
    BSL_AttrMap key = {
        .name = attrName
    } ;

    return (BSL_AttrMap const *const)
            bsearch(&key, class->attrs, class->attrCount,
            sizeof(key), bslAttrMapCompare) ;
}
```

```
<<harness forward function declarations>>=
static int bslAttrMapCompare(void const*, void const*) ;
```

```
<<harness static function definitions>>=
static int
bslAttrMapCompare(
    void const *m1,
```

```
    void const *m2)
{
    BSL_AttrMap const *cm1 = m1 ;
    BSL_AttrMap const *cm2 = m2 ;

    return strcmp(cm1->name, cm2->name) ;
}
```

Finding a Class Instance

A class instance may be specified as either a number or, for initial instance, a name. When searching for a class instance, we first try to see if it is specified as a number and then fall back to names should the number conversion fail.

```
<<harness forward function declarations>>=
static int bslFindInst(BSL_ClassMap const *const class, char const *instName) ;
```

```
<<harness static function definitions>>=
static int
bslFindInst(
    BSL_ClassMap const *const class,
    char const *instName)
{
    // First scan to see if we got a number.
    int instId = -1 ;
    int converted = sscanf(instName, "%i", &instId) ;
    if (converted == 1) {
        if (instId < 0) {
            bslFormatToContext(CmdContext,
                    "instance id's must be non-negative: got, %i", instId) ;
        }
    } else {
        // Not a number, try to look up the name.
        BSL_InstMap key = {
            .name = instName
        } ;

        BSL_InstMap const *const instMap =
                bsearch(&key, class->insts, class->instCount,
                sizeof(key), bslInstMapCompare) ;
        if (instMap == NULL) {
            bslFormatToContext(CmdContext,
                    "unknown instance, \"%s\"", instName) ;
            instId = -1 ;
        } else {
            instId = instMap->id ;
        }
    }

    return instId ;
}
```

```
<<harness forward function declarations>>=
static int bslInstMapCompare(void const *m1, void const *m2) ;
```

```
<<harness static function definitions>>=
static int
bslInstMapCompare(
    void const *m1,
    void const *m2)
```

```
{
    BSL_InstMap const *cm1 = m1 ;
    BSL_InstMap const *cm2 = m2 ;

    return strcmp(cm1->name, cm2->name) ;
}
```

Finding a Class Event

```
<<harness forward function declarations>>=
static BSL_EventMap const *bslFindEvent(BSL_ClassMap const *const class,
        char const *eventName) ;
```

```
<<harness static function definitions>>=
static BSL_EventMap const *
bslFindEvent(
    BSL_ClassMap const *const class,
    char const *eventName)
{
    BSL_EventMap key = {
        .name = eventName
    } ;

    BSL_EventMap const *eventMap = bsearch(&key, class->events,
            class->eventCount, sizeof(key), bslEventMapCompare) ;

    if (eventMap == NULL) {
        bslFormatToContext(CmdContext, "unknown event, \"%s\"", eventName) ;
    }

    return eventMap ;
}
```

```
<<harness forward function declarations>>=
static int bslEventMapCompare(void const *m1, void const *m2) ;
```

```
<<harness static function definitions>>=
static int
bslEventMapCompare(
    void const *m1,
    void const *m2)
{
    BSL_EventMap const *cm1 = m1 ;
    BSL_EventMap const *cm2 = m2 ;

    return strcmp(cm1->name, cm2->name) ;
}
```

Finding a Class State

```
<<harness forward function declarations>>=
static BSL_StateMap const *bslFindState(BSL_ClassMap const *const class,
        char const *stateName) ;
```

```
<<harness static function definitions>>=
static BSL_StateMap const *
```

```
bslFindState(
    BSL_ClassMap const *const class,
    char const *stateName)
{
    BSL_StateMap key = {
        .name = stateName
    } ;

    BSL_StateMap const *stateMap = bsearch(&key, class->states,
            class->stateCount, sizeof(key), bslStateMapCompare) ;

    if (stateMap == NULL) {
        bslFormatToContext(CmdContext, "unknown state, \"%s\"", stateName) ;
    }

    return stateMap ;
}
```

```
<<harness forward function declarations>>=
static int bslStateMapCompare(void const *m1, void const *m2) ;
```

```
<<harness static function definitions>>=
static int
bslStateMapCompare(
    void const *m1,
    void const *m2)
{
    BSL_StateMap const *cm1 = m1 ;
    BSL_StateMap const *cm2 = m2 ;

    return strcmp(cm1->name, cm2->name) ;
}
```

Domain Dependent Code

In this section, we describe the Tcl code that generates the "C" code that is dependent upon the particulars of the domain. Previously, we presented "C" code which formed the common core of the test harness code and was independent of the details of the harnessed domain. That code, however, does depend upon data structures and functions that are specific to the details of the harnessed domain. For example, the code to locate a class in a domain is independent of the domain, but relies on generated data that is specific to the domain.

The approach to the code generation is same as that found in the code generation for `micca`, namely, a template is expanded. Command in the template are executed and the result of the embedded commands is substituted into the template. The embedded template commands query the domain model to find out the information needed to generate the correct code. Note, that the queries contained in the template commands reference the `micca` domain model. It will be useful to have that model readily available to understand the details of the queries being performed.

We start at the top level and then show the details of the various commands embedded in the template.

Generate Command

The `generate` command is the fundamental command to create the "C" code file for the test harness. Later we see how the `generate` procedure is used in the overall processing of the `bosal` program.

```
<<bosal commands>>=
proc generate {options} {
    <<generate: set up template expansion>>
```

```
    <<generate: generate code>>
}
```

Template expansion is accomplished using the `expander` package from *tcllib*.

```
<<required packages>>=
package require textutil::expander
```

```
<<generate: set up template expansion>>=
textutil::expander expand
expand setbrackets <% %>                              ; # ❶
expand errmode [dict get $options expanderror]
expand evalcmd "namespace eval [namespace current]::GenCode"     ; # ❷

variable GenCode::includeMain [dict get $options main]
```

❶ For convenience, we change the strings the expander uses to recognize an embedded command. This set is more convenient when dealing with "C" code.

❷ To make sure the command names resolve as we want, the expansion is done in a separate namespace.

```
<<generate: generate code>>=
try {
    variable codeTemplate
    return [expand expand $codeTemplate]
} on error {result opts} {
    # puts $::errorInfo
    return -options $opts $result
} finally {
    rename expand {}
}
```

Code Template

The template for the code generation is organized in the order required by the "C" compiler. This lets us insure that constructs are declared before they are defined as required by the language.

```
<<bosal data>>=
variable codeTemplate [textutil::adjust::undent {
/*
<<edit warning>>
<<copyright info>>

This file generated by bosal version:
<<version info>>
*/

/*
 * Include Files
 */
#define _POSIX_C_SOURCE 201112L
#include "bosal.h"
#include "micca_rt.h"
#include "micca_rt_internal.h"
/*
 * Domain Header Files
 */
<%includeDomainHeaders%>
```

```
#include <sys/socket.h>
#include <sys/time.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <setjmp.h>
#include <ctype.h>
#include <time.h>
#include <strings.h>

/*
 * Macros
 */
#ifndef NDEBUG
#   define  dbgprintf(...)    printf(__VA_ARGS__)
#else
#   define  dbgprintf(...)
#endif /* NDEBUG */

/*
 * Special characters used in parsing commands.
 */
#define ASCII_NUL       '\0'
#define BEGIN_QUOTE     '{'
#define END_QUOTE       '}'
#define ESCAPE_CHAR     '\\'

/*
 * The state of the command parser.
 */
#define OUTOFWORD   0
#define INWORD      1
#define INESCAPE    2
#define INQUOTE     3

/*
 * Constants
 */
<<harness constants>>

/*
 * Data Types
 */
<<harness data types>>

/*
 * Forward References
 */
<<harness forward function declarations>>

/*
 * Static Data
 */
<<harness static data>>
<%generateStateMaps%>
<%generateEventParams%>
<%generateEventMaps%>
<%generateInstMaps%>
<%generateAttrMaps%>
<%generateClassMaps%>
```

```
<%generateOpParams%>
<%generateOpMaps%>
<%generateHarnesses%>

/*
 * Static Functions
 */
<<harness static function definitions>>

/*
 * External Functions
 */
<<harness external function definitions>>
<%generateMain%>
}]
```

In the following sections, we show the Tcl code that is executed during the template expansion.

Generating main

Upon request, `bosal` will generate a `main()` function into the test harness code file. For simple cases, this main is sufficient to run the test harness and removes the burden of having yet another file in the project. The following Tcl procedure creates the code for the minimal `main()` function.

```
<<bosal code template commands>>=
proc generateMain {} {
    variable includeMain

    if ($includeMain) {
        set div [diverter new]                    ; # ❶

        $div puts int
        $div puts main( ; $div push
        $div puts "int argc,"
        $div puts "char **argv)" ; $div pop
        $div puts "{" ; $div push
        $div puts "mrt_Initialize() ;"
        $div puts "bsl_Run() ;"
        $div puts "return 0 ;" ; $div pop
        $div puts "}"

        $div return
    }

    return
}
```

❶ The Tcl code to generate the test harness code uses the concept of a text diversion to deal with indentation levels, etc.

Including the domain header files

The domain header files are derived from the name of a domain. This matches the naming convension used by `micca` in producing the header file.

```
<<bosal code template commands>>=
proc includeDomainHeaders {} {
    set div [diverter new]
```

```
    forAllRefs domainRef [Domain findAll] {
        set domainName [readAttribute $domainRef Name]
        $div puts "#include \"$domainName.h\""
    }

    $div return
}
```

Generating domain mappings

A major part of the link between the domain independent code and the harnessed domains is a set of mapping tables. The mapping tables typically map string values onto some internal reference into the domain. For example, knowing how to execute a domain operation involves mapping the name of the operation to a function that ultimately invokes the "C" function that is the domain operation provided by the model translation.

The mapping tables are ordered alphabetically by the string key. This allows us to use `bsearch` to find an entry. The tables are not very large, so binary search is sufficient for our purposes and easier to manage than a hash table.

Generating the domain harness

The top level mapping locates information about a domain and is a mapping from the name of the domain to its descriptive data.

The following data structure describes an harnessed domain.

```
<<harness data types>>=
typedef struct bsldomainharness {
    MRT_DomainPortal const *portal ;                    // ❶
    struct bsloperationmap const *const operations ;    // ❷
    unsigned operationCount ;                           // ❸
    struct bslclassmap const *const classes ;           // ❹
    unsigned classCount ;                               // ❺
} BSL_DomainHarness ;
```

❶ A pointer to the portal for the domain. The portal is needed to use the `micca` run-time portal functions to control aspects of the domain.

❷ A pointer to an array of descriptors for the domain operations of the domain. The domain operation descriptors are defined below.

❸ The number of elements in the `operations` array.

❹ A pointer to an array of descriptors for the classes in the domain. The class descriptors are defined below.

❺ The number of elements in the `classes` array.

The data for the set of domains in the harness is contained in an initialized array variable of file static scope.

We spend some effort here to describe how the generation of the harness mapping data structure happens. Other map generating procedures follow a similar pattern, so understanding this one will help in understanding the remaining ones.

```
<<bosal code template commands>>=
proc generateHarnesses {} {
    set div [diverter new]

    <<generateHarnesses: query micca model>>
    <<generateHarnesses: iterate over query result>>

    $div return
}
```

Much of the code generated by `bosal` contains data obtained from the population of the platform specific model as gathered by `micca`. When `bosal` runs, it obtains the domain data by reading the file saved by `micca` when the domain is translated. The relvars of the `micca` model reside in the `micca` namespace and so we can use those values directly.

Since the harness map needs information about the operations of a domain and the classes in the domain, we start by obtaining that information.

```
<<generateHarnesses: query micca model>>=
set domops [relation project $::micca::DomainOperation Domain Name]
set classes [relation project $::micca::Class Domain Name]
```

Here we projected out only the attributes of **DomainOperation** and **Class** that we intend to use. This just keeps things easier to grasp by removing unused attributes.

The main query follows. The desired result is a relation value that gives the domain name and two additional attributes giving the required information about domain operations and classes.

```
<<generateHarnesses: query micca model>>=
set domains [pipe {
    relation project $::micca::Domain Name |
    relation rename ~ Name Domain |
    rvajoin ~ $domops Ops |
    rvajoin ~ $classes Classes
}]
```

This query is a pipeline of four commands:

1. Start with the **Domain**s of the model and, again for simplicity, project out the name as the only attribute we are interested in.
2. Rename the **Name** attribute to be **Domain**. This makes the attribute have the same name as it has in the **DomainOperation** and **Class** relvars.
3. Perform a relational outer join against the domain operations. The relational outer join avoids **NULL** values in the result by creating a relation valued attribute containing the joined tuples. If there are no matching tuples, then the relation valued attribute is empty (*i.e.* it contains no tuples). The new relation valued attribute is called **Ops**.
4. Do the same thing with the classes, placing the result in an attribute named, **Classes**.

The result of the query has the following tabular heading. The tuples in the relation have three attribute: **Domain**, **Ops**, and **Classes**. There is one tuple in the relation for each domain in the harness.

```
+------+--------------------------+-------------------------+
|Domain|Ops                       |Classes                  |
|string|Relation                  |Relation                 |
+------+--------------------------+-------------------------+
|      |+------------------------+|+---------------------+  |
|      ||Name                    |||Name                 |  |
|      ||string                  |||string               |  |
|      |+------------------------+|+---------------------+  |
|      ||                        |||                      |  |
|      |+------------------------+|+---------------------+  |
+------+--------------------------+-------------------------+
```

Recall that the **Ops** and **Classes** attributes are also relations. These relations contain a single attribute, **Name** which is the name of the domain operation or class. There is one tuple in the **Ops** relation value for each domain operation in the domain and one tuple in the **Classes** for each class in the domain.

Now the array can be defined and initialized by iterating over the tuples in the result relation.

```
<<generateHarnesses: iterate over query result>>=
$div puts "static BSL_DomainHarness const bslHarnesses\[] = {"  ;   # ❶
$div push

relation foreach domain $domains -ascending Domain {
    relation assign $domain                                      ;   # ❷

    set opsCount [relation cardinality $Ops]
    set ops [expr {$opsCount > 0 ? "${Domain}__OPMAP" : "NULL"}] ;   # ❸

    set classCount [relation cardinality $Classes]
    set classes [expr {$classCount > 0 ? "${Domain}__CLASSMAP" : "NULL"}]

    $div puts "{" ; $div push

    $div puts ".portal = &${Domain}__PORTAL,"                    ;   # ❹
    $div puts ".operations = $ops,"
    $div puts ".operationCount = $opsCount,"
    $div puts ".classes = $classes,"
    $div puts ".classCount = $classCount"

    $div pop ; $div puts -nonewline "},"
}

$div pop ; $div puts "} ;"

$div puts "static int const bslHarnessCount =\
        [relation cardinality $domains] ;"                       ;   # ❺
```

❶ Define an array named `bslHarnesses` to hold the harness information. This will be followed by initializers for all the elements of the array.

❷ `relation assign` splits out the attributes into ordinary Tcl variables with the same name. Since **Ops** and **Classes** were relation valued, the corresponding Tcl variables will hold relation values.

❸ Here we see the use of a naming convention to tie together the harness information with the array of domain operation descriptors. Another embedded macro command will generate the information for the domain operations and will use this naming convention for the variable. Since in "C" the name of an array degrades to a pointer to the first element of the array, the simple array name gives us the pointer value we need here. We use `NULL` to indicate that there are no domain operations for the domain. The same strategy is also applied to the classes of the domain.

❹ Now the structure member initializers of each array element are written to the diversion that is used to accumulate the output returned as the result of the embedded macro command expansion.

❺ We generate a constant to know the number of elements in the harness information array.

Generating the operations map

Domain operation names are mapped to a corresponding function. That function has a fixed signature and serves as a wrapper to parse the arguments to the domain operation and then eventually, if there is no argument parsing error, invoke the actual domain operation. The wrapper function are automatically generated from the signature of the domain operations.

The wrapper function has the following prototype.

```
<<harness data types>>=
typedef bool BSL_DomainOpFunc(int, char *const *) ;
```

The mapping is from a string name to the wrapper function pointer and the descriptors of the domain operation parameters.

```
<<harness data types>>=
typedef struct bsloperationmap {
    char const *name ;
    BSL_DomainOpFunc *func ;
    struct bslopparams const *params ;
    size_t paramCount ;
} BSL_OperationMap ;
```

Following the same pattern as above, generating the domain operations map involves querying the model and then iterating over the results of the query.

```
<<bosal code template commands>>=
proc generateOpMaps {} {
    set div [diverter new]

    <<generateOpMaps: query micca model>>
    <<generateOpMaps: iterate over query result>>

    $div return
}
```

The model query involves the domain operations and any parameters to the operation.

```
<<generateOpMaps: query micca model>>=
set params [relation project $::micca::DomainOperationParameter\
        Domain Operation Name]
set domainops [pipe {
    relation project $::micca::DomainOperation Domain Name |
    relation rename ~ Name Operation |
    rvajoin ~ $params Params |
    relation group ~ Ops Operation Params
}]
```

This query results in a relation value with the following tabular heading.

```
+------+------------------------------+
|Domain|Ops                           |
|string|Relation                      |
+------+------------------------------+
|      |+-----------+--------+       |
|      ||Operation  |Params  |       |
|      ||string     |Relation|       |
|      |+-----------+--------+       |
|      ||           |+------+|       |
|      ||           ||Name  ||       |
|      ||           ||string||       |
|      ||           |+------+|       |
|      ||           ||      ||       |
|      ||           |+------+|       |
|      |+-----------+--------+       |
+------+------------------------------+
```

In words, the relation resulting from the query of the `micca` model relates the domain name to a relation valued attribute describing the domain operations. The operation relation value maps the operation name to another relation value attribute describing its parameters. The parameters relation gives the name, ordinal number and data type of the parameter.

```
<<generateOpMaps: iterate over query result>>=
relation foreach domainop $domainops {
    relation assign $domainop
```

```
    <<generateOpMaps: generate operations descriptions>>
}
```

Iterating over the domain operations allows us to create the initialized array variable that serves as the domain operations map.

```
<<generateOpMaps: generate operations descriptions>>=
$div puts "static BSL_OperationMap const ${Domain}__OPMAP\[] = {"

$div push
relation foreach op $Ops -ascending Operation {
    relation assign $op

    set paramCount [relation cardinality $Params]
    set paramMap [expr {$paramCount != 0 ?
            "${Domain}_${Operation}__OPPARAMS" : "NULL"}]

    $div puts "{" ; $div push
    $div puts ".name = \"$Operation\","
    $div puts ".func = ${Domain}_${Operation}__OPFUNC,"
    $div puts ".params = $paramMap,"
    $div puts ".paramCount = $paramCount"
    $div pop ; $div puts -nonewline "},"
}

$div pop
$div puts "} ;"
```

Generating the domain operation functions

We need forward declarations for all the domain operation functions to keep the compiler happy.

```
<<harness forward function declarations>>=
<%generateOpFuncDeclarations%>
```

```
<<bosal code template commands>>=
proc generateOpFuncDeclarations {} {
    set div [diverter new]

    forAllRefs dopRef [DomainOperation findAll] {
        assignAttribute $dopRef Domain Name
        $div puts "static bool ${Domain}_${Name}__OPFUNC(int, char *const *) ;"
    }

    $div return
}
```

As mentioned previously, the domain operation function is a wrapper to parse the string arguments obtained from the command into the internal forms needed by the actual domain operation.

```
<<harness static function definitions>>=
<%generateOpFuncDefinitions%>
```

```
<<bosal code template commands>>=
proc generateOpFuncDefinitions {} {
    set div [diverter new]

    <<generateOpFuncDefinitions: query micca model>>
    <<generateOpFuncDefinitions: iterate over query result>>
```

```
    $div return
}
```

The query is similar to the one used for the domain operation information in the harness.

```
<<generateOpFuncDefinitions: query micca model>>=
set domainops [pipe {
    relation project $::micca::DomainOperation Domain Name ReturnDataType |
    relation rename ~ Name Operation |
    rvajoin ~ $::micca::DomainOperationParameter Params
}]
```

The query generates a relation value with the following heading.

```
+------+--------------------+---------------+----------------------------+
|Domain|Operation           |ReturnDataType |Params                      |
|string|string              |string         |Relation                    |
+------+--------------------+---------------+----------------------------+
|      |                    |               |+------+------+--------+    |
|      |                    |               ||Name  |Number|DataType|    |
|      |                    |               ||string|int   |string  |    |
|      |                    |               |+------+------+--------+    |
|      |                    |               |+------+------+--------+    |
+------+--------------------+---------------+----------------------------+
```

We iterate over the tuples in the query result which gives us the information to invoke a single domain operation.

```
<<generateOpFuncDefinitions: iterate over query result>>=
relation foreach domainop $domainops {
    <<generateOpFuncDefinitions: generate operation function>>
}
```

Here we generate the domain operation wrapper function that is referenced in the domain operation mapping table. The structure of these functions is to:

1. Check the argument count to insure that it is correct.
2. Scan the arguments converting them to internal form.
3. Invoke the domain operation provided by the domain itself.

```
<<generateOpFuncDefinitions: generate operation function>>=
relation assign $domainop

$div puts "static bool ${Domain}_${Operation}__OPFUNC(" ; $div push ; # ❶
$div puts "int argc,"
$div puts "char *const *argv)" ; $div pop
$div puts "{" ; $div push

<<generateOpFuncDefinitions: check argument count>>

relation foreach param $Params -ascending Number {
    <<generateOpFuncDefinitions: scan argument value>>
}

<<generateOpFuncDefinitions: generate operation invocation>>

$div puts "return true ;"
$div pop ; $div puts "}"
```

❶ The wrapper functions all have the same signature. Naming convention is used to insure uniqueness of the function names.

We know the proper number of arguments to the domain operation since it is the same as the cardinality of the parameter relation value.

```
<<generateOpFuncDefinitions: check argument count>>=
set nparams [relation cardinality $Params]
$div puts "if (argc != $nparams) {" ; $div push
$div puts "bslFormatToContext(CmdContext, wrongNumArgsMsg, argc, $nparams) ;"
$div puts "return false ;"
$div pop ; $div puts "}"
```

Scanning the arguments implies we iterate over the parameters calling `sscanf()`. The parameters to `sscanf` are based on the type of the argument.

```
<<generateOpFuncDefinitions: scan argument value>>=
relation assign $param {DataType type} {Name paramName} {Number argIndex}
set typeinfo [GetTypeInfo $Domain $type $paramName]
$div puts "[dict get $typeinfo declaration] ;"

set scnfmt [dict get $typeinfo scan]
set varref [dict get $typeinfo varref]
if {$scnfmt ne {}} {
    $div puts "if (sscanf(argv\[$argIndex\], \"%\" $scnfmt,\
        $varref) != 1) {" ; $div push
    $div puts "bslFormatToContext(CmdContext, badParamMsg, argv\[$argIndex\]) ;"
    $div puts "return false ;" ; $div pop
    $div puts "}"
} else {
    $div puts "$paramName = argv\[$argIndex\] ;"
}
```

The domain operation is invoked using the converted argument values. If there is a return value, then it must be formatted into a string for the command response.

```
<<generateOpFuncDefinitions: generate operation invocation>>=
set invokeParams [pipe {
    relation list $Params Name -ascending Number |
    join ~ {, }
}]
set invocation "${Domain}_${Operation}($invokeParams) ;"

if {$ReturnDataType eq "void"} {
    $div puts $invocation
} else {
    set rettypeinfo [GetTypeInfo $Domain $ReturnDataType result__OP]
    set decl [dict get $rettypeinfo declaration]
    $div puts "$decl = $invocation ;"
    $div puts "bslFormatToContext(CmdContext,\
                \"%\" [dict get $rettypeinfo print], result__OP) ;"
}
```

Many of the wrapper functions generated have the same error messages. So, we declare them once and use them in the wrappers.

```
<<harness static data>>=
static char const wrongNumArgsMsg[] = "wrong # arguments: got %d, expected %d" ;
static char const badParamMsg[] = "bad parameter: \"%s\"" ;
```

Generating the operations parameter data

To support querying the name and data type of the operation parameters, we also need to record that information.

```
<<harness data types>>=
typedef struct bslopparams {
    char const *name ;          // ❶
    char const *dataType ;      // ❷
} BSL_DomainOpParam ;
```

❶ The name of the domain operation parameter.

❷ The "C" data type of the parameter.

The implementation of this procedure follows the established pattern. We make a query on a populated `micca` platform model and then use the result to generate "C" variables and their initializers.

```
<<bosal code template commands>>=
proc generateOpParams {} {
    set div [diverter new]

    set paramGrps [relation group $::micca::DomainOperationParameter Params\
            Name Number DataType]

    relation foreach paramGrp $paramGrps {
        relation assign $paramGrp

        $div puts "static BSL_DomainOpParam const\
            ${Domain}_${Operation}__OPPARAMS\[] = {"
        $div push

        relation foreach param $Params -ascending Number {
            relation assign $param

            $div puts "{" ; $div push
            $div puts ".name = \"$Name\","
            $div puts ".dataType = \"$DataType\""
            $div pop ; $div puts -nonewline "},"
        }

        $div pop ; $div puts "} ;"
    }

    $div return
}
```

Generating the class map

The class mapping data is more extensive since classes are associated with many other components. The following is the structure of the generated data.

```
<<harness data types>>=
typedef struct bslclassmap {
    char const *name ;                      // ❶
    MRT_ClassId id ;                        // ❷
    struct bslattrmap const *attrs ;        // ❸
    unsigned attrCount ;                    // ❹
    struct bslinstmap const *insts ;        // ❺
    unsigned instCount ;                    // ❻
```

```
    struct bsleventmap const *events ;      // ❼
    unsigned eventCount ;                   // ❽
    struct bslstatemap const *states ;      // ❾
    unsigned stateCount ;                   // ❿
} BSL_ClassMap ;
```

❶ The name of the class.

❷ A numeric ID for the class.

❸ A pointer to an array of attribute descriptions.

❹ The number of elements in the `attrs` array.

❺ A pointer to an array of instance descriptions.

❻ The number of elements in the `insts` array.

❼ A pointer to an array of event descriptions.

❽ The number of elements in the `events` array.

❾ A pointer to an array of state descriptions.

❿ The number elements in the `states` array.

To generate the class mapping data, we use a set of naming conventions for the variables that hold each of the components, *e.g.* attributes, instances, etc., and reference them when generating the initialized array that is the class map. Although we present the code generation for the class map first, followed by the code for the generation of the components, the template is ordered so as to expand the generation of the components first. This saves us from having to generate forward references for the component variables just to satisfy the compiler.

```
<<bosal code template commands>>=
proc generateClassMaps {} {
    set div [diverter new]

    # ❶
    set attrs [pipe {
        relation project $::micca::Attribute Domain Class Name |
        relation rename ~ Name AttrName
    }]
    set insts [relation project $::micca::ClassInstance Domain Class Instance]
    set states [pipe {
        relation project $::micca::StatePlace Domain Model Name |
        relation rename ~ Model Class Name StateName |
        relation restrictwith ~ {$StateName ne "@"}
    }]  ; # ❷
    set events [pipe {
        relation project $::micca::Event Domain Model Event |
        relation rename ~ Model Class Event EventName
    }]

    set domclasses [pipe {
        relation rename $::micca::Class Name Class Number ClassId |
        rvajoin ~ $attrs Attrs |
        rvajoin ~ $insts Insts |
        rvajoin ~ $states States |
        rvajoin ~ $events Events |
        relation group ~ Comps Attrs Insts States Events |
        relation group ~ Classes Class ClassId Comps
    }]  ; # ❸
```

```
    relation foreach domclass $domclasses {
        relation assign $domclass
        $div puts "static BSL_ClassMap const ${Domain}__CLASSMAP\[] = {"
        $div push

        relation foreach class $Classes -ascending Class {
            relation assign $class
            relation assign $Comps

            # ❹
            set attrCount [relation cardinality $Attrs]
            set attrMap [expr {$attrCount > 0 ?\
                "${Domain}_${Class}__ATTRMAP" : "NULL"}]

            set instCount [relation cardinality $Insts]
            set instMap [expr {$instCount > 0 ?\
                "${Domain}_${Class}__INSTMAP" : "NULL"}]

            set eventCount [relation cardinality $Events]
            set eventMap [expr {$eventCount > 0 ?\
                "${Domain}_${Class}__EVENTMAP" : "NULL"}]

            set stateCount [relation cardinality $States]
            set stateMap [expr {$stateCount > 0 ?\
                "${Domain}_${Class}__STATEMAP" : "NULL"}]

            # ❺
            $div puts "{" ; $div push
            $div puts ".name = \"$Class\","
            $div puts ".id = $ClassId,"
            $div puts ".attrs = $attrMap,"
            $div puts ".attrCount = $attrCount,"
            $div puts ".insts = $instMap,"
            $div puts ".instCount = $instCount,"
            $div puts ".events = $eventMap,"
            $div puts ".eventCount = $eventCount,"
            $div puts ".states = $stateMap,"
            $div puts ".stateCount = $stateCount,"
            $div pop ; $div puts -nonewline "},"
        }

        $div pop ; $div puts "} ;"
    }

    $div return
}
```

❶ We start by projecting out only the attributes of the components of the class that we need. Sometimes attributes also need to be renamed to make the join operations easier to specify.

❷ We remove any pseudo-initial states from the states of a class since that state is always transitory and used internally for asychronous instance creation.

❸ This series of `rvajoin` invocations creates a relation valued attribute for each of the components in which we are interested. With some grouping, we can obtain the classes for a domain with the class components in a convenient attribute form.

❹ Having built up relation valued attributes for each of the class components we are interested in, we visit each component to determine any subordinate mappings.

❺ Having all the required information in variables, we can now generate the “C” code to initialize the array member.

Generating the attribute map

The required mapping for a class attribute maps the attribute id value to a pair of functions, one for reading the attribute and one for writing the attribute. The following data structure declarations define the interface to the attribute access functions.

```
<<harness data types>>=
typedef bool BSL_AttrReadFunc(MRT_DomainPortal const *, MRT_ClassId,
        MRT_InstId, MRT_AttrId) ;
typedef bool BSL_AttrUpdateFunc(MRT_DomainPortal const *, MRT_ClassId,
        MRT_InstId, MRT_AttrId, char const *) ;

typedef struct bslattrmap {
    char const *name ;                  // ❶
    MRT_AttrId id ;                     // ❷
    BSL_AttrReadFunc *readFunc ;        // ❸
    BSL_AttrUpdateFunc *updateFunc ;    // ❹
} BSL_AttrMap ;
```

❶ The name of the attribute.

❷ A numerical identifier for the attribute.

❸ A pointer to a function to read the attribute value.

❹ A pointer to a function to update the attribute value.

```
<<bosal code template commands>>=
proc generateAttrMaps {} {
    set div [diverter new]

    set attrGrps [pipe {
        relation project $::micca::Attribute Domain Class Name |
        relation group ~ Attrs Name
    }]

    relation foreach attrGrp $attrGrps {
        relation assign $attrGrp
        $div puts "static BSL_AttrMap const ${Domain}_${Class}__ATTRMAP\[] = {"
        $div push

        set attrNum -1
        relation foreach attr $Attrs -ascending Name {
            relation assign $attr
            $div puts "{" ; $div push

            $div puts ".name = \"$Name\","
            $div puts ".id = [incr attrNum],"
            $div puts ".readFunc = ${Domain}_${Class}_${Name}__RD,"
            $div puts ".updateFunc = ${Domain}_${Class}_${Name}__UP"

            $div pop ; $div puts -nonewline "},"
        }

        $div pop ; $div puts "} ;"
    }

    $div return
}
```

Generating the attribute read functions

For the attribute read functions, we will need forward declarations for the function signatures.

```
<<harness forward function declarations>>=
<%generateAttrFuncDeclarations%>
```

Note that we simply place the embedded macro command that generates the forward declarations into the appropriate literate program chunk. We use this technique in other situations that follow. Also note that we have combined the forward declaration for the read and update functions. Both use the same information from the `micca` model and forward declarations are not order dependent.

```
<<bosal code template commands>>=
proc generateAttrFuncDeclarations {} {
    set div [diverter new]

    relation foreach attr $::micca::Attribute {
        relation assign $attr Domain Class Name
        $div puts "static bool\
            ${Domain}_${Class}_${Name}__RD(MRT_DomainPortal const *,\
            MRT_ClassId, MRT_InstId, MRT_AttrId) ;"
        $div puts "static bool\
            ${Domain}_${Class}_${Name}__UP(MRT_DomainPortal const *,\
            MRT_ClassId, MRT_InstId, MRT_AttrId, char const *) ;"
    }

    $div return
}
```

The attribute read function is just an ordinary function at file static scope.

```
<<harness static function definitions>>=
<%generateAttrReadFuncDefinitions%>
```

```
<<bosal code template commands>>=
proc generateAttrReadFuncDefinitions {} {
    set div [diverter new]

    relation foreach attr $::micca::Attribute {
        relation assign $attr Domain Class Name DataType

        $div puts "static bool ${Domain}_${Class}_${Name}__RD(" ; $div push
        $div puts "MRT_DomainPortal const *portal,"
        $div puts "MRT_ClassId classId,"
        $div puts "MRT_InstId instId,"
        $div puts "MRT_AttrId attrId)" ; $div pop

        $div puts "{" ; $div push

        set typeinfo [GetTypeInfo $Domain $DataType attrValue]
        set attrVar [dict get $typeinfo declaration]

        $div puts "$attrVar ;"

        set pAttrVar [dict get $typeinfo varref]

        $div puts "int pcode = mrt_PortalReadAttr(portal, classId, instId,\
                attrId, $pAttrVar, sizeof(attrValue)) ;"
        $div puts "if (pcode < 0) {" ; $div push
        $div puts "bslResetContext(CmdContext) ;"
        $div puts "bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;"
```

```
        $div puts "return false ;"
        $div pop ; $div puts "}"

        set varType [dict get $typeinfo type]
        set prifmt [dict get $typeinfo print]
        switch -exact -- $varType {
            scalar {
                $div puts "bslFormatToContext(CmdContext, \"%\" $prifmt, attrValue) ;"
            }
            string {
                $div puts "bslFormatToContext(CmdContext, \"%\" $prifmt, attrValue) ;"
            }
            array {
                set dim [dict get $typeinfo dimension]
                $div puts "for (int index = 0 ; $index < $dim ; index++) {"
                $div push
                $div puts "bslFormatToContext(CmdContext, \"%\" $prifmt \" \",\
                        attrValue\[index]) ;"
                $div pop ; $div puts "}"
            }
            default {
                $div puts "bslResetContext(CmdContext) ;"
                $div puts "bslFormatToContext(CmdContext,
                    \"unknown variable type, $varType\") ;"
                $div puts "return false ;"
            }
        }

        $div puts "return true ;"

        $div pop ; $div puts "}"
    }

    $div return
}
```

Generating the attribute update functions

The generation of the attribute update functions follows the same pattern as for the attribute read functions.

```
<<harness static function definitions>>=
<%generateAttrUpdateFuncDefinitions%>
```

```
<<bosal code template commands>>=
proc generateAttrUpdateFuncDefinitions {} {
    set div [diverter new]

    relation foreach attr $::micca::Attribute {
        relation assign $attr Domain Class Name DataType

        $div puts "static bool ${Domain}_${Class}_${Name}__UP(" ; $div push
        $div puts "MRT_DomainPortal const *portal,"
        $div puts "MRT_ClassId classId,"
        $div puts "MRT_InstId instId,"
        $div puts "MRT_AttrId attrId,"
        $div puts "char const *value)" ; $div pop

        $div puts "{" ; $div push

        $div puts "int pcode ;"
```

```
        set typeinfo [GetTypeInfo $Domain $DataType convertedValue]
        switch -exact -- [dict get $typeinfo type] {
            scalar {
                $div puts "[dict get $typeinfo declaration] ;"
                set cvtRef [dict get $typeinfo varref]

                if {[dict get $typeinfo basetype] eq "bool"} {
                    $div puts "int nitems = bslScanBoolean(value, $cvtRef) ;"
                    set scnfmt \"bool\"
                } else {
                    set scnfmt [dict get $typeinfo scan]
                    $div puts "int nitems = sscanf(value, \"%\" $scnfmt,\
                            $cvtRef) ;"
                }
                $div puts "if (nitems == 1) {" ; $div push
                $div puts "pcode = mrt_PortalUpdateAttr(portal, classId,\
                    instId, attrId, $cvtRef, sizeof(convertedValue)) ;"
                $div pop ; $div puts "} else {" ; $div push
                $div puts "bslFormatToContext(CmdContext, failedConvertMsg,\
                        value, $scnfmt) ;"
                $div puts "return false ;"
                $div pop ; $div puts "}"
            }
            string {
                $div puts "pcode = mrt_PortalUpdateAttr(portal, classId,\
                    instId, attrId, value, strlen(value) + 1) ;"
            }
            array {
                $div puts "bslFormatToContext(CmdContext,
                    \"unable to update array attributes\") ;"
            }
        }

        $div puts "if (pcode < 0) {" ; $div push
        $div puts "bslFormatToContext(CmdContext, mrt_PortalErrorString(pcode)) ;"
        $div puts "return false ;"
        $div pop ; $div puts "}"

        $div puts "return true ;"

        $div pop ; $div puts "}"
    }

    $div return
}
```

```
<<harness static data>>=
static char const failedConvertMsg[] = "failed to convert, \"%s\" as \"%s\"" ;
```

Generating the instance map

The instance mapping is from an instance name to an instance ID. Note that only initial instances have a name.

```
<<harness data types>>=
typedef struct bslinstmap {
    char const *name ;
    MRT_InstId id ;
} BSL_InstMap ;
```

```
<<bosal code template commands>>=
proc generateInstMaps {} {
    set div [diverter new]

    set instGrps [relation group $::micca::ClassInstance Insts Instance Number]

    relation foreach instGrp $instGrps {
        relation assign $instGrp {Domain domainName} {Class className} Insts

        $div puts "static BSL_InstMap const\
                ${domainName}_${className}__INSTMAP\[] = {"
        $div push

        relation foreach inst $Insts -ascending Instance {
            relation assign $inst {Instance instName} {Number instNum}
            $div puts "{" ; $div push

            $div puts ".name = \"$instName\","
            $div puts ".id = $instNum,"

            $div pop ; $div puts -nonewline "},"
        }

        $div pop ; $div puts "} ;"
    }

    $div return
}
```

Generating the event map

The event mapping data maps the event name to an event number. The additional complication is that some events carry supplimental argument data. For those cases, a function is generated to convert the event argument data to internal form and additional descriptive data is generated to support querying the event parameters.

```
<<harness data types>>=
typedef bool BSL_EventParamFunc(int, char * const*, MRT_EventParams) ;

typedef struct bsleventmap {
    char const *name ;
    MRT_EventCode number ;
    BSL_EventParamFunc *paramFunc ;
    struct bsleventparams const *params ;
    size_t paramCount ;
} BSL_EventMap ;
```

One complication here is that we must deal with polymorphic events and assigner events. We do *not* allow access to assigner events, so they must be excluded. However, we do allow polymorphic events so they can be signaled to superclasses. However, a superclass is not required to have a state model. Polymorphic events do not affect any state model that the superclass may have. So our query for which events to generate mappings is slightly more complicated. We must start at instance state models (to eliminate the assigners) to find any events those state models may have. Also we must start at superclasses and find all the deferred events they may be related to and these will be polymorphic events for those superclasses (either inherited or directly defined). The union is then the set of events for which we are interested in generating maps. Since we need this set of events in several places, we create a helper function to do the query.

```
<<bosal helper commands>>=
proc GetMappedEvents {} {
    set instevents [pipe {
        InstanceStateModel findAll |
```

```
        findRelated % R50 ~R87 R80 |
        deRef %
    } {} |%]

    set defevents [pipe {
        Superclass findAll |
        findRelated ~ R86 R80 |
        deRef ~
    }]

    return [relation union $instevents $defevents]
}
```

```
<<bosal code template commands>>=
proc generateEventMaps {} {
    set div [diverter new]

    set params [relation project $::micca::Parameter Domain PSigID Name]

    set eventGrps [pipe {
        GetMappedEvents |
        rvajoin ~ $params Params |
        relation eliminate ~ PSigID |
        relation group ~ Events Event Number Params
    }]
    # puts [relformat $eventGrps eventGrps]

    relation foreach eventGrp $eventGrps {
        relation assign $eventGrp

        $div puts "static BSL_EventMap const ${Domain}_${Model}__EVENTMAP\[] = {"
        $div push

        relation foreach event $Events -ascending Event {
            relation assign $event
            set nparams [relation cardinality $Params]
            if {$nparams > 0} {
                set paramStore ${Domain}_${Model}_${Event}__EVPARAM
                set paramFunc ${Domain}_${Model}_${Event}__EPFUNC
            } else {
                set paramStore NULL
                set paramFunc NULL
            }

            $div puts "{" ; $div push

            $div puts ".name = \"$Event\","
            $div puts ".number = $Number,"
            $div puts ".paramFunc = $paramFunc,"
            $div puts ".params = $paramStore,"
            $div puts ".paramCount = $nparams"

            $div pop ; $div puts -nonewline "},"
        }

        $div pop ; $div puts "} ;"
    }

    $div return
}
```

Generating the event parameter functions

Following a similar pattern as attribute read and update functions, we generate event parameter functions. This include forward declarations and the conversion functions themselves.

```
<<harness forward function declarations>>=
<%generateEventParamFuncDeclarations%>
```

```
<<bosal code template commands>>=
proc generateEventParamFuncDeclarations {} {
    set div [diverter new]

    set events [pipe {
        GetMappedEvents |
        relation restrictwith ~ {$PSigID ne {}}
    }]
    # puts [relformat $events events]

    relation foreach event $events {
        relation assign $event {Domain domainName} {Model className}\
            {Event eventName}
        $div puts "static bool\
            ${domainName}_${className}_${eventName}__EPFUNC(\
            int, char * const*, MRT_EventParams) ;"
    }

    $div return
}
```

To generate the functions used to convert the event parameter values to the internal representation, we must include data type information in what we retrieve about the event parameters.

```
<<harness static function definitions>>=
<%generateEventParamFuncDefinitions%>
```

```
<<bosal code template commands>>=
proc generateEventParamFuncDefinitions {} {
    set div [diverter new]

    set paramtypes [pipe {
        relation join $::micca::Parameter $::micca::Argument |
        relation eliminate ~ ASigID
    }]

    set paramevents [pipe {
        GetMappedEvents |
        rvajoin ~ $paramtypes Params |
        relation eliminate ~ PSigID |
        relation restrictwith ~ {[relation isnotempty $Params]}
    }]
    # puts [relformat $paramevents paramevents]

    relation foreach paramevent $paramevents {
        relation assign $paramevent

        $div puts "static bool ${Domain}_${Model}_${Event}__EPFUNC(" ; $div push
        $div puts "int paramc,"
        $div puts "char * const *paramv,"
        $div puts "MRT_EventParams parambuf)" ; $div pop

        $div puts "{" ; $div push
```

```
        set nparams [relation cardinality $Params]
        $div puts "if (paramc != $nparams) {" ; $div push
        $div puts "bslFormatToContext(CmdContext, wrongNumArgsMsg, paramc,\
                $nparams) ;"
        $div puts "return false ;"
        $div pop ; $div puts "}"

        set structName ${Domain}_${Model}_${Event}__EPARAMS
        $div puts "struct $structName *params =\
                (struct $structName *)parambuf ;"

        set nitemsDefined false
        relation foreach param $Params -ascending Position {
            relation assign $param

            set typeinfo [GetTypeInfo $Domain $DataType params->$Name]
            set scnfmt [dict get $typeinfo scan]
            if {$scnfmt eq {}} {
                $div puts "params->$Name = paramv\[$Position\] ;"
            } else {
                if {!$nitemsDefined} {
                    $div puts "int nitems ;"
                    set nitemsDefined true
                }
                set reftovar [dict get $typeinfo varref]
                if {[dict get $typeinfo basetype] eq "bool"} {
                    $div puts "nitems = bslScanBoolean(paramv\[$Position\],\
                            $reftovar) ;"
                    set scnfmt \"bool\"
                } else {
                    $div puts "nitems = sscanf(paramv\[$Position\],\
                        \"%\" $scnfmt, $reftovar) ;"
                }
                $div puts "if (nitems != 1) {" ; $div push
                $div puts "bslFormatToContext(CmdContext, failedConvertMsg,\
                        paramv\[$Position\], $scnfmt) ;"
                $div puts "return false ;"
                $div pop ; $div puts "}"
            }
        }

        $div puts "return true ;"
        $div pop ; $div puts "}"
    }

    $div return
}
```

Generating the event parameter descriptions

The event parameter mapping maps an event name to its corresponding “C” data type.

```
<<harness data types>>=
typedef struct bsleventparams {
    char const *name ;          // ❶
    char const *dataType ;      // ❷
} BSL_EventParam ;
```

❶ The name of the event.

❷ The “C” data type of the event.

```
<<bosal code template commands>>=
proc generateEventParams {} {
    set div [diverter new]

    set params [pipe {
        relation join $::micca::Parameter $::micca::Argument |
        relation eliminate ~ ASigID
    }]

    set paramevents [pipe {
        GetMappedEvents |
        relation eliminate ~ Number |
        rvajoin ~ $params Params |
        relation eliminate ~ PSigID |
        relation restrictwith ~ {[relation isnotempty $Params]}
    }]
    # puts [relformat $paramevents paramevents]

    relation foreach paramevent $paramevents {
        relation assign $paramevent

        $div puts "static BSL_EventParam const\
            ${Domain}_${Model}_${Event}__EVPARAM\[] = {"
        $div push

        relation foreach param $Params -ascending Position {
            relation assign $param Name DataType

            $div puts "{" ; $div push
            $div puts ".name = \"$Name\","
            $div puts ".dataType = \"$DataType\""
            $div pop ; $div puts -nonewline "},"
        }

        $div pop ; $div puts "} ;"
    }

    $div return
}
```

Generate the state map

A state map gives the correspondence between the name of a state and its internal state number.

```
<<harness data types>>=
typedef struct bslstatemap {
    char const *name ;          // ❶
    MRT_StateCode number ;      // ❷
} BSL_StateMap ;
```

❶ The name of the state.

❷ The internal ID number of the state.

```
<<bosal code template commands>>=
proc generateStateMaps {} {
    set div [diverter new]

    set stateGrps [pipe {
        InstanceStateModel findAll |
        findRelated % R50 ~R55 R57 |
        deRef % |
        relation restrictwith % {$Name ne "@"} |
        relation group % States Name Number
    } {} |%]

    relation foreach stateGrp $stateGrps {
        relation assign $stateGrp

        $div puts "static BSL_StateMap const ${Domain}_${Model}__STATEMAP\[] = {"
        $div push

        relation foreach state $States -ascending Name {
            relation assign $state

            $div puts "{" ; $div push
            $div puts ".name = \"$Name\","
            $div puts ".number = $Number,"
            $div pop ; $div puts -nonewline "},"
        }

        $div pop ; $div puts "} ;"
    }

    $div return
}
```

Bosal header file

We must also be prepared to generate a header file for the harness that contains the interface information for the `bosal` external functions. This file must be included in the compilation unit that contains the `main()` function at the least. The `bosal` program accepts an option to create this header file.

```
<<bosal data>>=
variable headerFile [textutil::adjust::undent {
/*
<<edit warning>>
<<copyright info>>
*/
/*
 * This file is the header file for bosal. A typical use case is to
 * include it in the file which contains the "main" function. This
 * file corresponds to bosal version:
   <<version info>>
 */

#ifndef BOSAL_H_
#define BOSAL_H_

#define BOSAL_VERSION\
<<version info>>

#include <stdarg.h>
```

```
/*
 * External Functions
 */
<<harness external function declarations>>

#endif /* BOSAL_H_ */}]
```

Utility Code

The code in this section implements a set of utility functions used by `bosal` generated test harnesses.

Trimming White Space

Since responses in the communciations protocol are whitespace separated words, it is easier to generate a trailing space character at the end of each word added to a response. After adding the final word, this space can be trimmed off. This is simplier and avoids having to know which word added to a response is the first one. The following function performs the trimming.

```
<<harness forward function declarations>>=
static void bslTrimSpaceRight(char *const buf) ;
```

```
<<harness static function definitions>>=
static void
bslTrimSpaceRight(
    char *const buf)
{
    for (char *end = buf + strlen(buf) - 1 ;
            end > buf && isspace(*end) ; end--) {
        *end = ASCII_NUL ;
    }
}
```

Response Context

To help build up responses, we define the notion of a *response context*. A response context is a set of buffers that can be used to accumulate output in a nested fashion. Since the responses generated by `bosal` are grouped into *words* that are separated by whitespace, it convenient to have distinct areas into which formatted output can be generated before deciding if the output contains embedded white space and must be *quoted* to make it a single word. In this way, when words are added to a context after the formatting is accomplished it is much easier to determine if the new word contains characters that must be quoted.

A context is defined as a character buffer, a current location in the buffer where the next character is to be placed, and the number of bytes remaining in the buffer.

```
<<harness data types>>=
typedef struct bslbufcontext {
    char buf[BUFSIZ] ;
    char *location ;
    size_t remaining ;
} BSL_BufContext ;
```

A context stack is simply an array of buffer contexts with a pointer to the current position in the stack.

The stacking operations are implemented in an array of buffer contexts by manipulating the pointer to the current position. We need to define the depth of the context stack that will be supported. Static allocation is sufficient here since we know what depth responses require.

```
<<harness constants>>=
#ifndef BSL_NESTLIMIT
#   define  BSL_NESTLIMIT   4   /* maximum depth of text diversions */
#endif /* BSL_NESTLIMIT */
```

So a context stack consists of the storage for the buffer contexts and a pointer to the buffer context that is the current one.

```
<<harness data types>>=
typedef struct bslcontextstack {
    struct bslbufcontext *current ;
    struct bslbufcontext stack[BSL_NESTLIMIT] ;
} BSL_ContextStack ;
```

There are four different types of responses generated by a test harness.

```
<<harness data types>>=
typedef enum {
    CmdContext = 0,
    TraceContext,
    InstrContext,
    FatalContext,
    NUMContexts         // must be last
} BSL_ContextId ;
```

We define an array of context stack data structures to use as the context for each type of response from the test harness.

```
<<harness static data>>=
static BSL_ContextStack responseContexts[NUMContexts] ;
```

The `bslBeginResponse()` function is used to initialize the context in preparation for generated a response.

```
<<harness forward function declarations>>=
static void bslBeginResponse(BSL_ContextId cntx) ;
```

```
<<harness static function definitions>>=
static void
bslBeginResponse(
    BSL_ContextId cntx)
{
    assert(cntx < NUMContexts) ;
    if (cntx >= NUMContexts) {
        fprintf(stderr, "%s: invalid response context number, \"%d\"\n",
                __func__, cntx) ;
        abort() ;
    }
    BSL_ContextStack *stack = &responseContexts[cntx] ;
    stack->current = stack->stack ;                    // ❶
    bslInitBufContext(stack->current) ;
}
```

❶ Beginning a response causes any other partial response to be abandoned as we reset the current context to be the bottom of the stack.

A response is ended by invoking `bslEndResponse()`.

```
<<harness forward function declarations>>=
static int bslEndResponse(BSL_ContextId cntx) ;
```

```
<<harness static function definitions>>=
static inline int
bslEndResponse(
    BSL_ContextId cntx)
{
    assert(cntx < NUMContexts) ;
    if (cntx >= NUMContexts) {
        fprintf(stderr, "%s: invalid response context number, \"%d\"\n",
                __func__, cntx) ;
        abort() ;
    }

    BSL_ContextStack *stack = &responseContexts[cntx] ;

    while (stack->current > stack->stack) {
        bslContextStackPop(stack) ;                    // ❶
    }
    bslTrimBufContextRight(stack->current) ;

    return fprintf(bslCmdStream, "%s\n", stack->current->buf) ;
}
```

❶ Ending a response performs an implicit `pop` of the entire context stack.

When a new word is being formulated in a response, the context is pushed using `bslPushContext()`. Future response text is then placed in a new context so that it can be later evaluated to determine if it is necessary to quote the text to maintain word boundaries. Thus context pushes correspond to word boundaries in the response.

```
<<harness forward function declarations>>=
static void bslContextStackPush(BSL_ContextStack *stack) ;
static void bslPushContext(BSL_ContextId cntx) ;
```

```
<<harness static function definitions>>=
static void
bslContextStackPush(
    BSL_ContextStack *stack)
{
    assert(stack != NULL) ;
    assert(stack->current < stack->stack + BSL_NESTLIMIT) ;

    if (stack->current >= stack->stack + BSL_NESTLIMIT) {
        fprintf(stderr, "%s: buffer context stack overflow\n", __func__) ;
        abort() ;
    }
    stack->current++ ;
    bslInitBufContext(stack->current) ;
}

static void
bslPushContext(
    BSL_ContextId cntx)
{
    assert(cntx < NUMContexts) ;
    if (cntx >= NUMContexts) {
        fprintf(stderr, "%s: invalid response context number, \"%d\"\n",
                __func__, cntx) ;
        abort() ;
    }
```

```
    BSL_ContextStack *stack = &responseContexts[cntx] ;
    bslContextStackPush(stack) ;
}
```

When a word context is completed, it is popped using `bslPopContext()`. This adds any word accumulated in the context to the accumulated response, quoting things as necessary to adhere to the protocol.

```
<<harness forward function declarations>>=
static void bslContextStackPop(BSL_ContextStack *stack) ;
static void bslPopContext(BSL_ContextId cntx) ;
```

```
<<harness static function definitions>>=
static void
bslContextStackPop(
    BSL_ContextStack *stack)
{
    assert(stack != NULL) ;
    assert(stack->current > stack->stack) ;

    if (stack->current <= stack->stack) {
        fprintf(stderr, "buffer context stack underflow\n") ;
        abort() ;
    }

    char *currBuf = stack->current->buf ;
    bslTrimBufContextRight(stack->current) ;
    stack->current-- ;
    bsl_AddWord(stack->current, currBuf) ;
}

static void
bslPopContext(
    BSL_ContextId cntx)
{
    assert(cntx < NUMContexts) ;
    if (cntx >= NUMContexts) {
        fprintf(stderr, "%s: invalid response context number, \"%d\"\n",
                __func__, cntx) ;
        abort() ;
    }

    BSL_ContextStack *stack = &responseContexts[cntx] ;
    bslContextStackPop(stack) ;
}
```

A context may also be reset. This is useful in those cases where a response is being built up and it is necessary to clear things out because of an exceptional condition.

```
<<harness forward function declarations>>=
static inline void bslResetContext(BSL_ContextId cntx) ;
```

```
<<harness static function definitions>>=
static inline void
bslResetContext(
    BSL_ContextId cntx)
{
    assert(cntx < NUMContexts) ;
    if (cntx >= NUMContexts) {
        fprintf(stderr, "%s: invalid response context number, \"%d\"\n",
                __func__, cntx) ;
        abort() ;
```

```
    }

    BSL_ContextStack *stack = &responseContexts[cntx] ;
    bslInitBufContext(stack->current) ;
}
```

Responses are built up by adding words to the context.

```
<<harness forward function declarations>>=
static inline void bslAddWordToContext(BSL_ContextId cntx, char const *word) ;
```

```
<<harness static function definitions>>=
static inline void
bslAddWordToContext(
    BSL_ContextId cntx,
    char const *word)
{
    assert(cntx < NUMContexts) ;
    if (cntx >= NUMContexts) {
        fprintf(stderr, "%s: invalid response context number, \"%d\"\n",
                __func__, cntx) ;
        abort() ;
    }

    BSL_ContextStack *stack = &responseContexts[cntx] ;
    bsl_AddWord(stack->current, word) ;
}
```

Initializing a buffer context sets all the bookkeeping back to the beginning of the buffer.

```
<<harness forward function declarations>>=
static void bslInitBufContext(BSL_BufContext *bufc) ;
```

```
<<harness static function definitions>>=
static void
bslInitBufContext(
    BSL_BufContext *bufc)
{
    assert(bufc != NULL) ;

    bufc->location = bufc->buf ;
    *bufc->location = ASCII_NUL ;
    bufc->remaining = sizeof(bufc->buf) ;
}
```

Adding words means we must surround it by braces if it needs to be quoted.

```
<<harness forward function declarations>>=
static int bsl_AddWord(BSL_BufContext *bufc, char const *word) ;
```

```
<<harness static function definitions>>=
static int
bsl_AddWord(
    BSL_BufContext *bufc,
    char const *word)
{
    int nbytes = snprintf(bufc->location, bufc->remaining,
            bslQuoteWord(word) ? "{%s} " : "%s ", word) ;

    if (nbytes < 0 || nbytes >= bufc->remaining) {
        return -1 ;
```

```
    } else {
        bufc->remaining -= nbytes ;
        bufc->location += nbytes ;
    }

    return nbytes ;
}
```

When a context is popped, trailing whitespace is removed.

```
<<harness forward function declarations>>=
static void bslTrimBufContextRight(BSL_BufContext *bufc) ;
```

```
<<harness static function definitions>>=
static void
bslTrimBufContextRight(
    BSL_BufContext *bufc)
{
    while (bufc->location > bufc->buf) {
        if (isspace(*(bufc->location - 1))) {
            *--bufc->location = ASCII_NUL ;
        } else {
            break ;
        }
    }
}
```

Finally, the heavy lifting is done by the formatted I/O functions for a context.

```
<<harness forward function declarations>>=
static int bslVformatToContext(BSL_ContextId cntx, char const *format,
        va_list args) ;
```

```
<<harness static function definitions>>=
static int
bslVformatToContext(
    BSL_ContextId cntx,
    char const *format,
    va_list args)
{
    assert(cntx < NUMContexts) ;
    if (cntx >= NUMContexts) {
        fprintf(stderr, "%s: invalid response context number, \"%d\"\n",
                __func__, cntx) ;
        abort() ;
    }

    BSL_ContextStack *stack = &responseContexts[cntx] ;
    BSL_BufContext *bufc = stack->current ;

    int nbytes = vsnprintf(bufc->location, bufc->remaining, format, args) ;

    if (nbytes < 0 || nbytes >= bufc->remaining) {
        return -1 ;
    } else {
        bufc->remaining -= nbytes ;
        bufc->location += nbytes ;
    }

    return nbytes ;
}
```

```
<<harness forward function declarations>>=
static int bslFormatToContext(BSL_ContextId cntx, char const *format, ...) ;
```

```
<<harness static function definitions>>=
static int
bslFormatToContext(
    BSL_ContextId cntx,
    char const *format,
    ...)
{
    va_list params ;
    va_start(params, format) ;
    int nbytes = bslVformatToContext(cntx, format, params) ;
    va_end(params) ;

    return nbytes ;
}
```

The following function determines if a text string needs to be quoted in order to be considered a word. It accomplished this by scanning the string until it can resolve whether it contains embedded white space or embedded braces. If embedded brace characters are found, then scanning continues in an attempt to find matching pairs of braces.

```
<<harness forward function declarations>>=
static bool bslQuoteWord(char const *pWord) ;
```

```
<<harness static function definitions>>=
static bool
bslQuoteWord(
    char const *pWord)
{
    assert(pWord != NULL) ;
    if (strlen(pWord) == 0) {
        return true ; // must quote the empty string
    }

    enum {
        OutOfQuote,
        InQuote
    } quoteState = OutOfQuote ;
    int quoteCount = 0 ;

    for (char c = *pWord++ ; c != ASCII_NUL ; c = *pWord++) {
        if (c == '\\') {
            pWord++ ;
            continue ;
        }
        switch (quoteState) {
        case OutOfQuote:
            if (isspace(c)) {
                return true ;
            } else if (c == BEGIN_QUOTE) {
                quoteCount++ ;
                quoteState = InQuote ;
            }
            break;

        case InQuote:
            if (c == BEGIN_QUOTE) {
                quoteCount++ ;
            } else if (c == END_QUOTE) {
```

```
                assert(quoteCount > 0) ;
                if (--quoteCount == 0) {
                    quoteState = OutOfQuote ;
                }
            }
            break;
        }
    }
    return quoteCount == 0 ? false : true ;
}
```

Scanning for boolean values we treat as special. We want to accept the usual *true* and *false*, but also allow for numerical zero and one.

```
<<harness static function definitions>>=
static int bslScanBoolean(char const *value, bool *result) ;
```

```
<<harness static function definitions>>=
static int
bslScanBoolean(
    char const *value,
    bool *result)
{
    assert(value != NULL) ;
    assert(result != NULL) ;

    if (strcasecmp(value, "true") == 0) {
        *result = true ;
    } else if (strcasecmp(value, "false") == 0) {
        *result = false ;
    } else {
        char *end ;
        unsigned num = strtoul(value, &end, 0) ;
        if (*end == ASCII_NUL) {
            if (num == 0) {
                *result = false ;
            } else if (num == 1) {
                *result = true ;
            } else {
                return 0 ;
            }
        } else {
            return 0 ;
        }
    }

    return 1 ;
}
```

Helper Commands

In this section we present a set of commands that factor common processing used in a number of areas. These commands are placed in a separate namespace which is imported into most of the other package namespaces.

Text diversion

The first time I encountered the idea of a text diversion was in the venerable `m4` macro processor on UNIX. The concept is to be able to have multiple streams being accumulated and then later bring the diverted text back into the desired order.

For our usage here in generating code, we need a way to make code indentation easier to manage so that the generated code looks natural and as if hand coded. So, we employ the idea of a text diversion. When a new code block entered, we **push** the diversion as a stack. This causes subsequent text to be held separately. When the diversion is later **popped**, the text is placed into the next lowest stream, but indented to the proper level (given the number of previously occuring pushes).

In Tcl, this is all rather easy to accomplish since *tcllib* contains all the components we need. We will use in-memory FIFO channels to hold diverted text. We use a stack to keep track of the stack of file channel handles. Finally, indentation is handled by text utilities in *tcllib*. The code is formulated as a TclOO class called, `diverter`.

```
<<bosal helper commands>>=
package require tcl::chan::events
package require tcl::chan::fifo
package require struct::stack
package require oo::util
package require textutil::adjust

oo::class create diverter {
    variable chan_stack
    constructor {{prefix {    }}} {
        my variable level_prefix
        set level_prefix $prefix
        classvariable stack_counter                                ; # ❶
        set chan_stack [::struct::stack s_[incr stack_counter]]
        $chan_stack push [::tcl::chan::fifo]
    }
    destructor {
        foreach dchan [$chan_stack get] {
            chan close $dchan
        }
        $chan_stack destroy
    }
    method puts {args} {
        set nargs [llength $args]
        set curr_chan [$chan_stack peek]
        if {$nargs == 1} {
            chan puts $curr_chan [lindex $args 0]
        } elseif {$nargs == 2} {
            lassign $args opt str
            chan puts $opt $curr_chan $str
        } else {
            error "wrong # of arguments: expected, \"puts ?-nonewline? string\""
        }
    }
    method push {} {
        $chan_stack push [::tcl::chan::fifo]
    }
    method pop {} {
        set top_chan [$chan_stack pop]
        chan flush $top_chan

        set curr_chan [$chan_stack peek]
        my variable level_prefix
        set diverted [::textutil::adjust::indent\
                [chan read $top_chan] $level_prefix]
        chan puts $curr_chan $diverted

        chan close $top_chan
    }
    method undivert {} {
        while {[$chan_stack size] > 1} {
            my pop
        }
```

```
        set curr_chan [$chan_stack peek]
        chan flush $curr_chan
        return [chan read $curr_chan]
    }
    method return {} {
        set result [my undivert]
        my destroy
        return -level 2 $result         ; # ❷
    }
}
```

❶ Used to make unique names for stack component of the diverter.

❷ Note that the `return` method returns all the diverted text, destoys itself and returns out of the calling procedure. This is convenient since the diversions are used to end the procedures that are embedded in the template expansion and it results in all the accumulated output being placed into the expansion.

Data type procedures

We need some common code to process "C" data types from the `micca` domains.

Usually, we need to know the base "C" type of something. Since it is possible to create a type alias for a type alias, a bit of recursion is needed to resolve down to the underlying "C" data type.

```
<<bosal helper commands>>=
proc UnaliasType {domain typename} {
    set aliasRef [TypeAlias findById Domain $domain TypeName $typename]
    if {[isEmptyRef $aliasRef]} {
        return $typename
    } else {
        return [UnaliasType $domain [readAttribute $aliasRef TypeDefinition]]
    }
}
```

The code generator has to create conversion wrapper functions for "C" data types. This procedure summarizes the information needed by code generation. This area may need additional work.

```
<<bosal helper commands>>=
# return a dictionary keyed by:
# print -- format string
# scan -- format string
# basetype -- unaliased type
# type -- scalar, string or array
# base  -- base type of array or string if type is not scalar
# dimension  -- array or string dimension if type is not scalar
# declaration -- variable declaration with the provided variable name
# varref -- reference to the provided variable name

proc GetTypeInfo {domain implType varName} {
    set basetype [UnaliasType $domain $implType]
    set info [typeCheck assignmentType $basetype]
    dict set info basetype $basetype
    switch -exact -- [dict get $info type] {
        scalar {
            set fmt [relvar restrictone FormatMap ImplType $basetype]
            if {[relation isempty $fmt]} {
                dict set info print "\"i\""
                dict set info scan "\"i\""
            } else {
                dict set info print [relation extract $fmt Print]
```

```
                dict set info scan [relation extract $fmt Scan]
            }
            dict set info varref &$varName
        }
        string {
            dict set info print "\"s\""
            dict set info scan "\"s\""
            dict set info varref $varName
        }
        array {
            set fmt [relvar restrictone FormatMap ImplType\
                [dict get $info base]]
            if {[relation isempty $fmt]} {
                dict set info print "\"d\""
                dict set info scan "\"d\""
            } else {
                dict set info print [relation extract $fmt Print]
                dict set info scan [relation extract $fmt Scan]
            }
            dict set info varref $varName
        }
    }

    dict set info declaration [typeCheck composeDeclaration $basetype $varName]

    return $info
}
```

This is a simple mapping relation for how to format and scan “C” basic data types.

```
<<bosal helper data>>=
# Create a simple map that maps "C" implementation types onto
# printf/scanf format strings.
relvar create FormatMap {
    ImplType string
    Print string
    Scan string
} ImplType

relvar set FormatMap [relation table {
    ImplType string             Print string         Scan  string   } {
    int                         {"d"}                {"i"}               } {
    unsigned                    {"u"}                {"i"}               } {
    {unsigned int}              {"u"}                {"i"}               } {
    short                       {"hd"}               {"hi"}              } {
    {short int}                 {"hd"}               {"hi"}              } {
    {unsigned short}            {"hu"}               {"hi"}              } {
    {unsigned short int}        {"hu"}               {"hi"}              } {
    long                        {"ld"}               {"li"}              } {
    {long int}                  {"ld"}               {"li"}              } {
    {unsigned long}             {"lu"}               {"li"}              } {
    {unsigned long int}         {"lu"}               {"li"}              } {
    size_t                      {"z"}                {"z"}               } {
    {long long}                 {"lld"}              {"lli"}             } {
    {long long int}             {"lld"}              {"lli"}             } {
    {unsigned long long}        {"llu"}              {"lli"}             } {
    {unsigned long long int}    {"llu"}              {"lli"}             } {
    {char *}                    {"s"}                {}                  } {
    {char const *}              {"s"}                {}                  } {
    {const char *}              {"s"}                {}                  } {
    {void *}                    {"p"}                {"p"}               } {
    {const void *}              {"p"}                {"p"}               } {
```

```
    {void const *}              {"p"}               {"p"}               } {
    float                       {"g"}               {"g"}               } {
    double                      {"lg"}              {"lg"}              } {
    {long double}               {"Lg"}              {"Lg"}              } {
    char                        {"hhd"}             {"hhi"}             } {
    {unsigned char}             {"hhu"}             {"hhi"}             } {
    void                        void                {}                  } {
    bool                        {"u"}               {"u"}               } {
    uint8_t                     PRIu8               SCNi8               } {
    uint16_t                    PRIu16              SCNi16              } {
    uint32_t                    PRIu32              SCNi32              } {
    uint64_t                    PRIu64              SCNi64              } {
    int8_t                      PRId8               SCNi8               } {
    int16_t                     PRId16              SCNi16              } {
    int32_t                     PRId32              SCNi32              } {
    int64_t                     PRId64              SCNi64              } {
    intmax_t                    PRIdMAX             SCNiMAX             } {
    uintmax_t                   PRIuMAX             SCNiMAX             } {
    typename(time_t)            {"ld"}              {"li"}              } {
    MRT_ClassId                 {"hu"}              {"hi"}              } {
    MRT_AssignerId              {"hu"}              {"hi"}              } {
    MRT_InstId                  {"hu"}              {"hi"}              } {
    MRT_AttrId                  {"hu"}              {"hi"}              } {
    MRT_AttrSize                {"z"}               {"z"}               } {
    MRT_AttrOffset              {"lu"}              {"li"}              } {
    MRT_DelayTime               PRIu32              SCNi32              } {
    MRT_EventCode               PRIu8               SCNi8               } {
    MRT_StateCode               PRId8               SCNi8               } ]
```

Bosal Program Organization

In this section we show the organization of the files that can be tangled from the literate source.

Source Code

We start with some preliminaries. `Bosal` is implemented in Tcl. Since `Bosal` also uses data generated by the `micca` program, we need the `micca` package. Since `micca` is based on `rosea`, the `rosea` package will also be included, transitively.

```
<<required packages>>=
package require micca
package require textutil::adjust
```

```
<<bosal.tcl>>=
<<edit warning>>
<<copyright info>>

package require Tcl 8.6
package require logger
<<required packages>>

namespace eval ::bosal {
    variable version\
    <<version info>>

    logger::initNamespace [namespace current]

    proc logproc {level txt} {
        if {[string match {-_logger*} $txt]} {
```

```
            set txt [lindex $txt end]
        }
        puts "bosal: $level: $txt"
    }
    proc errorlogproc {txt} {
        logproc error $txt
    }
    proc warnlogproc {txt} {
        logproc warn $txt
    }
    log::logproc error [namespace current]::errorlogproc
    log::logproc warn [namespace current]::warnlogproc

    <<bosal data>>
    <<bosal commands>>

    namespace export generate
    namespace ensemble create

    namespace eval Helpers {
        namespace path [list\
            ::micca\
            ::rosea::InstCmds\
        ]

        namespace import ::ral::*
        namespace import ::ralutil::*
        namespace import ::micca::@Config@::Helpers::typeCheck

        <<bosal helper data>>
        <<bosal helper commands>>
    }
    namespace eval GenCode {
        namespace path [list\
            ::micca\
            ::rosea::InstCmds\
            [namespace parent]::Helpers\
        ]

        namespace import ::ral::*
        namespace import ::ralutil::*
        namespace import ::micca::@Config@::Helpers::typeCheck

        <<bosal code template commands>>
    }
}

package provide bosal\
<<version info>>
```

Bosal Starpack Application

Tcl applications can be packaged as a single file executable with no external dependencies. This is known as a “starpack”. When `bosal` is packaged as an application in a starpack, this code is the main entry point.

```
<<bosal_main.tcl>>=
<<edit warning>>
<<copyright info>>

set iswrapped [expr {[lindex [file system [info script]] 0] ne "native"}]
```

```
if {$iswrapped} {
    set libdir [file join $::starkit::topdir lib]
    set appdir [file join $libdir application]
    set libs [list]
    if {$::tcl_platform(os) eq "Linux"} {
        set libs [glob -nocomplain -directory $libdir P-linux*]
    } elseif {$::tcl_platform(os) eq "Darwin"} {
        set libs [glob -nocomplain -directory $libdir P-macosx*]
    }
    foreach lib $libs {
        lappend ::auto_path $lib
    }
} else {
    set appdir [file dirname [info script]]
}

source [file join $appdir bosal.tcl]
package require bosal
package require cmdline

set cmdOpts {
    {version {Print out version information and exit}}
    {header {Output a copy of the header file, bosal.h and exit}}
    {output.arg {-} {Output file name}}
    {main {Include a minimal main function}}
    {level.arg warn {Logger print level}}
    {nogenerate {Don't generate any output files}}
    {doc {Output a copy of the manual documentation and exit}}
    {expanderror.arg fail {How macro expansion errors are handled}}
}

set usage "\[options] file1 ...\noptions:\n"
try {
    set options [cmdline::getoptions argv $cmdOpts $usage]
} on error {result} {
    puts stderr $result
    exit 1
}

::bosal::log::setlevel [dict get $options level]

if {[dict get $options version]} {
    chan puts "bosal: version $::bosal::version"
    chan puts {
<<copyright info>>
}
    exit 0
} elseif {[dict get $options doc]} {
    file copy -force [file join $appdir HTML] ./bosaldoc
    exit 0
} elseif {[dict get $options header]} {
    set hchan [open bosal.h w]
    try {
        chan puts $hchan $::bosal::headerFile
    } on error {result opts} {
        return -options $opts $result
    } finally {
        chan close $hchan
    }
    exit 0
}
```

```
if {[llength $argv] == 0} {
    puts stderr [::cmdline::usage $cmdOpts $usage]
    exit 1
}

set nerrs 0
foreach file $argv {
    try {
        rosea restore ::micca $file
    } on error result {
        puts $::errorInfo
        incr nerrs
    }
}
if {$nerrs != 0} {
    exit $nerrs
}

if {![dict get $options nogenerate]} {
    try {
        set harness [bosal generate $options]
        set outfilename [dict get $options output]
        if {$outfilename eq "-"} {
            chan puts -nonewline $harness
        } else {
            set ochan [open $outfilename w]
            try {
                chan puts -nonewline $ochan $harness
            } on error {result opts} {
                return -options $opts $result
            } finally {
                chan close $ochan
            }
        }
    } on error {result} {
        puts stderr $result
        exit 1
    }
}

exit 0
```

Copyright Information

This software is copyrighted. It is licensed in the same manner as Tcl itself.

```
<<copyright info>>=
# This software is copyrighted 2017 - 2018 by G. Andrew Mangogna.
# The following terms apply to all files associated with the software unless
# explicitly disclaimed in individual files.
#
# The authors hereby grant permission to use, copy, modify, distribute,
# and license this software and its documentation for any purpose, provided
# that existing copyright notices are retained in all copies and that this
# notice is included verbatim in any distributions. No written agreement,
# license, or royalty fee is required for any of the authorized uses.
# Modifications to this software may be copyrighted by their authors and
# need not follow the licensing terms described here, provided that the
# new terms are clearly indicated on the first page of each file where
# they apply.
```

```
#
# IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
# OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
# THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
# IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
# NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
# OR MODIFICATIONS.
#
# GOVERNMENT USE: If you are acquiring this software on behalf of the
# U.S. government, the Government shall have only "Restricted Rights"
# in the software and related documentation as defined in the Federal
# Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
# are acquiring the software on behalf of the Department of Defense,
# the software shall be classified as "Commercial Computer Software"
# and the Government shall have only "Restricted Rights" as defined in
# Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
# the authors grant the U.S. Government and others acting in its behalf
# permission to use and distribute the software in accordance with the
# terms specified in this license.
```

Version Information

```
<<version info>>=
"1.2"
```

Edit Warning

```
<<edit warning>>=
# DO NOT EDIT THIS FILE!
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
```

Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the Tcl interpreter. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the Tcl code in an order suitable for the Tcl interpreter. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to the Tcl interpreter.

Index