NAME

bcg_write – a simple interface to produce a BCG graph

DESCRIPTION

This interface generates a BCG graph from an application program written in C or C++. To keep things simple, this interface does not give access to the whole BCG format, but only to a subset, in which states are assumed to be unsigned integer numbers and labels of the edges are assumed to be character strings. Note: this subset of BCG is equivalent to the **.aut** format described in the **aldebaran**(LOCAL) manual page, although it is much more compact.

USAGE

The application program should start with the following directive:

```
#include "bcg_user.h"
```

Then the BCG library should be initialized by invoking the following function:

```
BCG_INIT ();
```

Not invoking this function might cause a run-time error, e.g., a segmentation fault. Invoking `BCG_INIT()` more than once is harmless, although not recommended.

DATA TYPES

The functions of this interface use the followings types, whose definitions are provided by the "bcg_user.h" file:

- **BCG_TYPE_BOOLEAN**
- **BCG_TYPE_NATURAL**
- **BCG_TYPE_C_STRING**
- **BCG_TYPE_FILE_NAME**
- **BCG_TYPE_LABEL_STRING**
- **BCG_TYPE_STATE_NUMBER**
- **BCG_TYPE_DATA_FORMAT**

FEATURES

....

BCG_IO_WRITE_BCG_BEGIN

```
BCG_TYPE_BOOLEAN BCG_IO_WRITE_BCG_BEGIN (filename,
                 initial_state, format, comment, monitor)

  BCG_TYPE_FILE_NAME                    filename;
  BCG_TYPE_STATE_NUMBER                 initial_state;
  BCG_TYPE_NATURAL                      format;
  BCG_TYPE_C_STRING                     comment;
  BCG_TYPE_BOOLEAN                      monitor;
  { ... }
```

This function opens a BCG file. Its arguments have the following meaning:

filename

is a character string containing the path name of the BCG file to be written. It should contain the "`.bcg`" suffix (if the "`.bcg`" suffix is missing, it will be added automatically).

initial_state

is the number of the initial state (usually 0).

format is equal to 2 if, in the forthcoming successive invocations of function **BCG_IO_WRITE_BCG_EDGE()**, the sequence of actual values given to the *state1* argument of **BCG_IO_WRITE_BCG_EDGE()** will increase monotonically or equal to 1 otherwise. The format 1 applies in general but is less efficient in terms of time or BCG file compression. The format 2 is more efficient but only applies to specific situations (in particular, when the BCG graph is generated in a breadth-first search).

comment
is a character string containing information about the application tool which is creating the BCG graph. If *comment* is equal to NULL, then it will be replaced with a default string.

monitor should be equal to **BCG_TRUE** if a Tcl/Tk window should be opened for monitoring in real-time the generation of the BCG graph or equal to **BCG_FALSE** otherwise.

By default, if *filename* cannot be opened for writing, **BCG_IO_WRITE_BCG_BEGIN()** will emit an error message and exit the program. However, if the following function call:

```
BCG_IO_WRITE_BCG_SURVIVE (BCG_TRUE);
```

has occured before the call to **BCG_IO_WRITE_BCG_BEGIN()**, then **BCG_IO_WRITE_BCG_BEGIN()** will neither emit an error message nor exit the program, but return normally a boolean result that is equal to **BCG_TRUE** if and only if *filename* cannot be opened.

Below, we assume that the **BCG_IO_WRITE_BCG_BEGIN()** function has returned successfully.

....

BCG_IO_WRITE_BCG_SURVIVE

```
BCG_IO_WRITE_BCG_SURVIVE (mode)

  BCG_TYPE_BOOLEAN mode;
  { ... }
```

This function controls how the **BCG_IO_WRITE_BCG_BEGIN()** function defined above will behave if the BCG file cannot be opened for writing:

- If *mode* equals **BCG_FALSE**, then **BCG_IO_WRITE_BCG_BEGIN()** will emit an error message and exit the program. This is the default behaviour.
- If *mode* equals **BCG_TRUE**, then **BCG_IO_WRITE_BCG_BEGIN()** will neither emit an error message nor exit the program, but return a boolean result. The default behaviour can be restored by calling:

```
BCG_IO_WRITE_BCG_SURVIVE (BCG_FALSE);
```

....

BCG_IO_WRITE_BCG_EDGE

```
BCG_IO_WRITE_BCG_EDGE (state1, label, state2)

  BCG_TYPE_STATE_NUMBER state1;
  BCG_TYPE_LABEL_STRING label;
  BCG_TYPE_STATE_NUMBER state2;
  { ... }
```

This function must be invoked once for each edge to be created in the BCG graph. It writes in the previously opened BCG file an edge such that *state1* is the number of the origin state, *state2* is the number of the destination state, and *label* is a character string containing the label.

Note: *label* should not contain the characters newline (`'\n'`) or carriage return (`'\r'`).

Note: the invisible (also known as hidden, or tau) label is represented by the character string **`"i"`** (as it is the case in LOTOS).

....

`BCG_IO_WRITE_BCG_PARSING`

```
BCG_IO_WRITE_BCG_PARSING (data_format)

  BCG_TYPE_DATA_FORMAT data_format;
  { ... }
```

This function can be (optionally) invoked to modify the way labels are parsed. For details about label parsing, see the **LABEL PARSING** section below.

Calling **`BCG_IO_WRITE_BCG_PARSING()`** with *data_format* equal to the constant value **`BCG_UNPARSED_DATA_FORMAT`** disables the parsing of labels.

Calling **`BCG_IO_WRITE_BCG_PARSING()`** with *data_format* equal to the constant value **`BCG_STANDARD_DATA_FORMAT`** enables the parsing of labels.

By default (i.e., if **`BCG_IO_WRITE_BCG_PARSING()`** is not invoked), label parsing is enabled. In order to improve inter-operability between tools, we recommend to leave label parsing enabled.

The call to **`BCG_IO_WRITE_BCG_PARSING()`** must be done after invoking **`BCG_INIT()`** and before invoking **`BCG_IO_WRITE_BCG_BEGIN()`**.

....

`BCG_IO_WRITE_BCG_END`

```
BCG_IO_WRITE_BCG_END ()

  { ... }
```

This function properly closes the BCG file.

....

`BCG_IO_WRITE_BCG_ABORT`

```
BCG_IO_WRITE_BCG_ABORT ()

  { ... }
```

This function stops the generation of the BCG file, and removes this file. This function should be invoked in case of a fatal error, so as not to leave an unfinished, invalid BCG file.

....

EXAMPLE

The following piece of C code creates a BCG graph with an initial state numbered 0:

```
#include "bcg_user.h"
int main ()
{
  BCG_TYPE_STATE_NUMBER S1;
  BCG_TYPE_LABEL_STRING L;
  BCG_TYPE_STATE_NUMBER S2;

  BCG_INIT ();
  BCG_IO_WRITE_BCG_BEGIN ("test.bcg", 0, 2, "created by tool", 1);
  /* for each transition labelled with L from state S1 to state S2 */
  {
    BCG_IO_WRITE_BCG_EDGE (S1, L, S2);
  }
  BCG_IO_WRITE_BCG_END ();
  return (0);
}
```

LABEL PARSING

The *label* argument passed to the **`BCG_IO_WRITE_BCG_EDGE()`** function is a character string that should only contain printable characters; the meaning of "printable" is given by the POSIX isprint() function with locale "C" (namely, ASCII characters with decimal codes in the range from 32 to 126, bounds included). Otherwise, the effect is undefined. In particular, a label string should not be terminated with line-feed and/or carriage-return characters. Wide characters (e.g., UTF-8, UTF-16, etc.) are not supported because their usefulness for concurrency theory may not be worth their complexity.

The *label* arguments can be interpreted in two different ways, depending whether label parsing is enabled or not.

If label parsing is disabled, each label is stored in the generated BCG file as a 1-tuple (L) whose unique field L is exactly the character string passed to **`BCG_IO_WRITE_EDGE()`**. In the generated BCG file, this unique field has the RAW type (see the FIELD PARSING section below for information about this type).

Note: In legacy BCG files generated by CADP versions up to 2014-h included, this unique field had the STRING type; this situation is detected by more recent versions of CADP, which automatically convert that field to the RAW type to ensure backward compatibility.

The remainder of this section discusses the case where label parsing is enabled.

In such case, each label string is assumed to a notation for a tuple of typed data values (V0, V1, ..., V*n*) called *fields*. The number *n* is not necessarily the same for all labels, meaning that each label can have its own number of fields.

Therefore, function **`BCG_IO_WRITE_BCG_EDGE()`** will attempt at parsing its *label* argument to cut it into a sequence of fields, and analyze each field to infer its type and extract its value. In the resulting BCG file, labels will be represented as tuple of field values.

Parsing labels is not mandatory, but it is recommended, because it enables certain model checkers of CADP, namely **evaluator4**(LOCAL) and **xtl**(LOCAL) to know about the types present in labels and to use field values to express powerful temporal logic properties.

There are other CADP tools that treat labels as mere character strings, and do not attempt at considering fields individually. For such tools, label parsing can still be enabled, as it is (almost) transparent: each label string is parsed and stored in the BCG file as a tuple of binary fields; this tuple can later be converted back into a label string by those tools that do not examine fields. Notice, however, that this latter label string can be slightly different from the former one, because of various *normalization* actions that will be described below.

As a tribute to tradition, parsed labels are not noted as mathematical tuples: **"(V0, V1, V2, ..., Vn)"**, but use the notation for labels established by the CSP and LOTOS process algebras: **"V0 !V1 !V2 !... !Vn"**. If there is only one field, the label is noted **"V0"**.

It is recommended to leave one space before the '**!**' character and no space after. Labels that do not follow this convention will still be parsed properly, but will be normalized under the recommended form if converted back to character strings.

Rules for label parsing continue with the next section, which specifically discusses the parsing of label fields.

FIELD PARSING

Fields can be of eight possible types: GATE, BOOLEAN, NATURAL, INTEGER, REAL, CHARACTER, STRING, and RAW.

Note: BCG files generated with versions 1.0 or 1.1 of the BCG format (namely, before September 2014) had only six of these types. The NATURAL and RAW types were not defined; these files did not contain any value of the CHARACTER type, and their STRING type behaved as the current RAW type.

For some of these eight types, there can be multiple field notations that lead to the same value. For instance, the two different fields **"1"** and **"01"** express the same number. Normalization also applies to fields and puts them under canonical textual representation.

The rules for parsing fields are applied in the following order:

(1) If the first field V0 of a label starts with a letter that is followed by any number of letters, digits, and/or underscore characters, then it is recognized and inserted in the BCG file as a value of the GATE type. This type is an enumerated type that gathers all the gate identifiers contained in the labels of the graph. For each gate identifier, a corresponding constant function of type GATE is inserted in the BCG file. The precise rules for translating fields denoting GATE values into binary values stored in memory, and vice versa, are given by two functions **bcg_gate_scan()** and **bcg_gate_print()** defined in file **$CADP/incl/adt_gate.h**.

(2) If a field is equal to "**TRUE**" or "**FALSE**" (or to their case-insensitive variants "**true**", "**false**", "**True**", "**False**", etc.), then it is recognized and inserted in the BCG file as a value of the BOOLEAN type. The precise rules for translating fields denoting BOOLEAN values into binary values stored in memory, and vice versa, are given by two functions **bcg_boolean_scan()** and **bcg_boolean_print()** defined in file **$CADP/incl/adt_boolean.h**.

Normalization converts BOOLEAN values to upper-case letters, either "**`TRUE`**" or "**`FALSE`**".

(3) If a field denotes an unsigned integer number (e.g., **`0`**, **`1`**, **`9999`**, etc.), then it is recognized and inserted in the BCG file as a value of the NATURAL type. The syntax of NATURAL fields is the one accepted by the POSIX function **strtoul**(3). The number must not be prefixed with a '**`+`**' sign. At present, only 32-bit unsigned integers are recognized as values of the NATURAL type; larger numbers will be considered as values of the REAL type. The precise rules for translating fields denoting NATURAL values into binary values stored in memory, and vice versa, are given by two functions **`bcg_natural_scan()`** and **`bcg_natural_print()`** defined in file **`$CADP/incl/adt_natural.h`**. Normalization removes leading zeros in NATURAL values.

(4) If a field denotes a signed integer number (e.g., -9999, **`+0`**, **`+1`**, **`+9999`**, etc.), then it is recognized and inserted in the BCG file as a value of the INTEGER type. The syntax of INTEGER fields is the one accepted by the POSIX function **strtol**(3). The number (even if it is zero) must be prefixed with either a '**`+`**' or '**`-`**' sign. At present, only 32-bit signed integers are recognized as values of the INTEGER type; smaller or larger numbers will be considered as values of the REAL type. The precise rules for translating fields denoting INTEGER values into binary values stored in memory, and vice versa, are given by two functions **`bcg_integer_scan()`** and **`bcg_integer_print()`** defined in file **`$CADP/incl/adt_integer.h`**. Normalization removes leading zeros in INTEGER values.

(5) If a field denotes a floating-point number (e.g., **`3.1415`**, **`-1.2E+10`**, etc.), then it is recognized and inserted in the BCG file as a value of the REAL type. The syntax of REAL fields is the one accepted by the POSIX function **strtod**(3). The precise rules for translating fields denoting REAL values into binary values stored in memory, and vice versa, are given by two functions **`bcg_real_scan()`** and **`bcg_real_print()`** defined in file **`$CADP/incl/adt_real.h`**. Normalization removes leading zeros and may add trailing zeros to REAL values.

(6) If a field denotes a character value (e.g., **`'a'`**, **`'\012'`**, **`'\x0A'`**, **`'\n'`**, etc.), then it is recognized and inserted in the BCG file as a value of the CHARACTER type. The syntax of CHARACTER fields is a simplified subset of the C language syntax: characters are enclosed between single quotes; any printable character *c* different from the single quote and backslash characters can be used to form a value '*c*'; the octal notation '\ooo' (where *ooo* denotes exactly three octal digits) and the hexadecimal notation '\x*hh*' (where *hh* denotes exactly two hexadecimal digits) are supported; the standard C notations **`'\0'`**, **`'\a'`**, **`'\b'`**, **`'\f'`**, **`'\n'`**, **`'\r'`**, **`'\t'`**, **`'\v'`**, **`'\"'`**, **`'\\'`**, **`'\''`** and **`'\?'`** are also supported; notice that the single quote and backslash characters must be always be escaped with a backslash, whereas the double quote and interrogation mark characters may be escaped or not, e.g., **`'"'`** or **`'\"'`**. The precise rules for translating fields denoting CHARACTER values into binary values stored in memory, and vice versa, are given by two functions **`bcg_character_scan()`** and **`bcg_character_print()`** defined in file **`$CADP/incl/adt_character.h`**. Normalization converts any unprintable CHARACTER value (the meaning of "printable" being given by the POSIX isprint() function with locale "C") into three-digit octal notation; printable CHARACTER values are displayed as such; for instance, **`'\n'`** and **`'\x23'`** are normalized as **`'\012'`** and **`'#'`**, respectively.

(7) If a field denotes a string value (e.g., **`""`**, **`"a\012\x04\n"`**, etc.), then it is recognized and inserted in the BCG file as a value of the STRING type. The syntax of STRING fields is a subset of that of the C language: characters are enclosed between double quotes and follow the same notations as for the CHARACTER type. The only two differences are the following: single quotes (which are always escaped in characters) can be escaped or not in strings, while double quotes (which can be escaped or not in characters) must be escaped in strings. The precise rules for

translating fields denoting STRING values into binary values stored in memory, and vice versa, are given by two functions **`bcg_string_scan()`** and **`bcg_string_print()`** defined in file **`$CADP/incl/adt_string.h`**. Normalization follows the same rules as for the CHARACTER type. If a null character (noted '**`\0`**', '**`\000`**', or '**`\x00`**') occurs in the middle of a field, the remaining characters of this field will be ignored.

(8) If a field does not match any of the rules above (e.g., **`x+y`**, **`cons (1, nil)`**, etc.), then it is considered as "raw data" and inserted in the BCG file as a value of the RAW type. This type gathers all values whose type cannot be determined easily. In particular, any empty field is considered as a value of type RAW. A field of type RAW may also contain "embedded" character and string values (e.g., **`pair ("name", 'k', 28)`**, etc.). Characters in RAW fields follow the same notations as for the CHARACTER type. The '**`!`**' character, which usually signals the end of the current field and the beginning of a new field, may occur in a RAW field provided that it is escaped (i.e., noted '**`\!`**', '**`\041`**', or '**`\x21`**'); however, this character does not need to be escaped if it occurs in an embedded character or an embedded string. Values of the RAW type are stored in memory as byte strings. The precise rules for translating fields denoting RAW values into binary values stored in memory, and vice versa, are given by two functions **`bcg_raw_scan()`** and **`bcg_raw_print()`** defined in file **`$CADP/incl/adt_raw.h`**.

To summarize the effects of label parsing and normalization on one example, the following label string **`" G! true ! 003!+01E+0 "`** passed to function **`BCG_IO_WRITE_BCG_EDGE()`** will be stored in the BCG file as a tuple (G, TRUE, 3, 1.0). If converted back to a character string, it will display as "G !TRUE !3 !1.000000".

COMPILING AND LINK EDITING

To compile the application tool, the following options must be passed to the C or C++ compiler:

-I$CADP/incl -L$CADP/bin.'$CADP/com/arch' -lBCG_IO -lBCG -lm

as in, e.g.,

```
$CADP/src/com/cadp_cc tool.c -o tool -I$CADP/incl \
-L$CADP/bin.'$CADP/com/arch' -lBCG_IO -lBCG -lm
```

EXIT STATUS

Application tools share common conventions with respect to diagnostics. Exit status is 0 if everything is alright, 1 otherwise.

AUTHORS

Hubert Garavel (definition of the BCG format) and Renaud Ruffiot (implementation of the BCG environment).

FILES

See the **bcg**(LOCAL) manual page for a description of the files.

SEE ALSO

bcg(LOCAL)

Additional information is available from the CADP Web page located at http://cadp.inria.fr

Directives for installation are given in files **$CADP/INSTALLATION_∗.**

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY.**

BUGS

Please report bugs to Hubert.Garavel@inria.fr