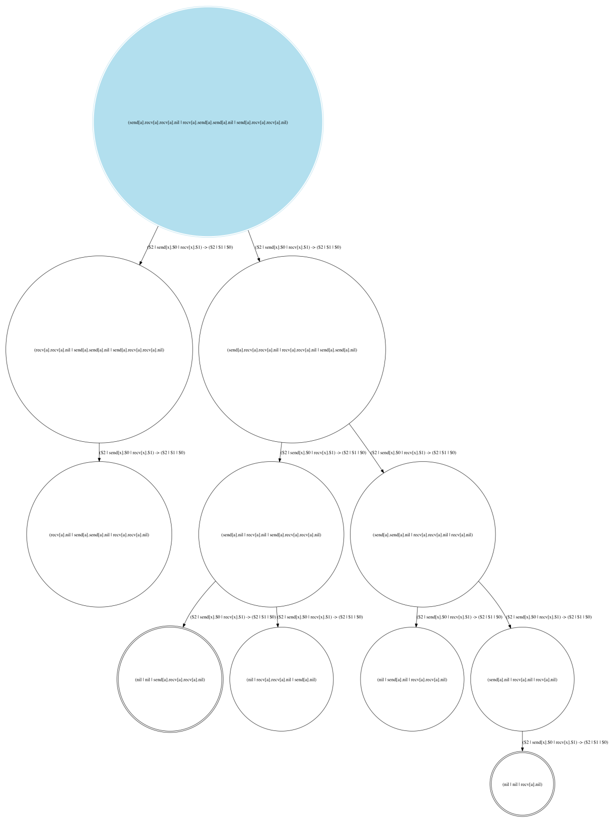
BigMC

for version 20110810


(send[a].recv[a].recv[a].nil | recv[a].send[a].send[a].nil | send[a].recv[a].recv[a].nil)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
(recv[a].recv[a].nil | send[a].send[a].nil | send[a].recv[a].recv[a].nil)
(send[a].recv[a].recv[a].nil | recv[a].recv[a].nil | send[a].send[a].nil)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
(recv[a].nil | send[a].send[a].nil | recv[a].recv[a].nil)
(send[a].nil | recv[a].nil | send[a].recv[a].recv[a].nil)
(send[a].send[a].nil | recv[a].recv[a].nil | recv[a].nil)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
(nil | nil | send[a].recv[a].recv[a].nil)
(nil | recv[a].recv[a].nil | send[a].nil)
(nil | send[a].nil | recv[a].recv[a].nil)
(send[a].nil | recv[a].nil | recv[a].nil)
($2 | send[x].$0 | recv[x].$1) -> ($2 | $1 | $0)
(nil | nil | recv[a].nil)

Gian Perrone (gdpe at itu.dk)

This work funded in part by the Danish Research Agency (grant no.: 2106-080046) and the IT University of Copenhagen (the Jingling Genies project)

Table of Contents

1 Introduction

BigMC (**Big**raphical **M**odel **C**hecker) is a model-checker designed to operate on *Bigraphical Reactive Systems* — BRS. BRS is a formalism developed by Robin Milner and colleagues that emphasises the orthogonal notions of *locality* and *connectivity*. Bigraphs have found applications in ubiquitous computing, computational biology, business workflow modelling, and context-aware systems.

By *model checking*, we mean precisely the act of checking whether some specification is true of a particular bigraphical model. This is achieved through a kind of exhaustive search of all the possible states of the system. For arbitrary models, this kind of checking is computationally intractible — the state space is simply too huge (and indeed infinite in many cases). The challenge of this kind of task is to limit the kinds of models we check to some tractible subset, and to reduce the number of actual states that we need to check directly in order to provide concrete correctness guarantees.

1.1 A cautionary note

BigMC is experimental software, still under active development. It might never be finished. It would be foolish to depend upon the kinds of guarantees that it can provide without considering exactly **why** these might be true.

1.2 About BigMC

BigMC is being developed at the IT University of Copenhagen by Gian Perrone in collaboration with Thomas Hildebrandt and Søren Debois.

2 Obtaining BigMC

The latest version of BigMC will always be available from the primary project website:

- Download BigMC-20110810 (tar.gz, latest as of 10 August 2011):
  `http://bigraph.org/bigmc/release/bigmc-20110810.tar.gz`

The latest version of this manual is available in PDF format from
`http://bigraph.org/bigmc/bigmc.pdf`
or in HTML format from:
`http://bigraph.org/bigmc/manual`.

2.1 Installation

BigMC uses a standard GNU Autoconf/Automake setup. You will need a Unix-like environment — the current release is known to work on MacOS X 10.6, MacOS X 10.7 and recent version of Debian and Ubuntu.

If you're lucky, the following procedure will install everything to `/usr/local/bigmc/`:

```
tar -xzf bigmc-20110810.tar.gz
cd bigmc-20110810/
./configure
make
sudo make install
```

The build process relies on the presence of the following:

- A C++ compiler (tested with The GNU C++ compiler `g++`, version 4.0 or greater)
- GNU Make
- GNU Readline

Highly recommended:

- Google-Perftools

Finally, it is recommended that you add something like the following to your '`.profile`':

```
export PATH=/usr/local/bigmc/bin:$PATH
export BIGMC_HOME=/usr/local/bigmc
```

All going well, you should be able to just invoke BigMC using the command:

```
$ bigmc
```

2.2 Configuration

The configuration file '`/usr/local/bigmc/conf/bigmc.conf`' contains predicate definitions that point to the dynamically loadable modules that implement them. If you wish to add new predicates, you will need to add them to this configuration file.

3 Invoking bigmc

Running `bigmc` alone will enter the interactive environment:

```
$ bigmc
BigMC version 0.1-dev (http://www.itu.dk/~gdpe/bigmc)
Copyright (c) 2011 Gian Perrone <gdpe at itu.dk>
bigmc>
```

The `bigmc>` prompt indicates that the interactive environment is ready to accept model definitions, properties and finally the `check` command.

To exit the interactive environment, enter `quit` or *C-d*. It is often more useful to invoke `bigmc` with a model file. The full command line options are:

```
bigmc [-hlpvV] [-G file] [-m steps] [-r steps] [-t threads] [file]
```

3.1 Command Line Options

Several options are available to control the runtime behaviour of the checking process:

`-h`	display usage information and exit.
`-l`	employ only local checking. This avoids building the transition system and saves memory when checking large models, but limits the properties that you can check to those relating only to a single agent.
`-v`	print version information and exit.
`-V -VV`	increase the verbosity of the information that is output during execution.
`-G` *file*	if set, output a graphviz file suitable for rendering with dot to *file*.
`-m` *steps*	set the maximum number of steps of reaction that may be performed.
`-p`	print new states as they are discovered.
`-r` *steps*	set the frequency with which statistics about the graph and work queue are output while running.
`-t` *threads*	instruct the checker to run *threads* parallel threads. Defaults to 2.

4 BGM file structure

The basic structure for a bigraph model file is as follows:

```
# Comments
<control definitions>

<names>

<reaction rules>

<model definition>

<properties>

%check;
```

4.1 Comments

Comments are lines starting with '#', and continue to the end of the line.

```
# This is a comment
```

4.2 Control Definitions

Control definitions take one of two forms:

```
%active control-name : arity;
%passive control-name : arity;
```

All of the information must be present. For example:

```
%passive send : 1;
%active foo : 3;
```

4.3 Names

The top-level outer names are defined as follows:

```
%name name;
```

For example:

```
%name a;
%name b;
```

The alternative keyword *%outer* is also accepted to make explicit the fact that these are specifically *outer* names.

Names need not be unique with respect to control names; however, it might be best to keep them that way to avoid confusion.

5 Term language

The term language is relatively simple. It uses the control names and inner and outer names as defined in the preamble. The full grammar is given by:

```
T ::= K.T
T ::= T | T
T ::= T || T
T ::= $n
T ::= K
T ::= nil

K ::= k[names]
K ::= k

names ::= n , names
names ::= n

n ::= [a-zA-Z][a-zA-Z0-9]*
n ::= -
```

Where *k* is drawn from the set of controls. Using `-` as a link name (e.g. `foo[-,x].P` leaves the first port of the *foo* node unlinked, and links the second port to *x*. Ports are linked in-order, so all ports must be present at every prefix, whether they are linked or not.

5.1 Reaction Rules

All terms of the form `T -> T;` are considered to be reaction rules for the model under construction. Order is not significant. For example:

```
send[x].$0 | recv[x].$1 -> $0 | $1;
```

Names that do not appear in `%name` definitions are considered free names and will be bound during matching. Names that are defined in the name set will be matched literally, and will not be re-bound.

The double-parallel operator (`||`) introduces new top-level regions. These can only appear at the top-level, and only in reaction rules. The number of regions in the redex and reactum must agree, e.g.:

```
a.$0 || b.$1 -> c.$0 || d.$0;
```

is a valid rule, however the following is not:

```
a.$0 || b.$1 -> c.$0;
```

A word of warning: The error detection for wide reaction rules is rather fragile in the current version. You would be well advised to be careful when constructing complex wide rules.

5.2 Model Definition

There can be at most one model definition per model file — this will always be the last line in the file of the form `T;`. It may appear anywhere in the file before the `%check` line, and after any definitions upon which it depends. Order with respect to reaction rules is not significant. Continuing from the previous sections, we could define a complete model:

```
%passive send : 1;
%passive recv : 1;
%name a;
%name b;

send[x].$0 | recv[x].$1 -> $0 | $1;

# The model definition
send[a].recv[b].send[a] | recv[a].send[b].recv[b];
```

We now have a complete model file suitable to use for checking.

6 Property language

Having defined a model in the previous section, we want to start to be able to make some kind of specification of properties which we would like to ensure hold. We do this using the `%property` specifier, which is of the form:

```
%property property-name property-expression;
```

The *property-name* is a non-semantically-significant name which will be reported if the checker encounters a violation of a property during checking.

A single '`.bgm`' file can have arbitrarily many `%property` declarations. The order in which they appear in the file will be the order in which they are considered at each new state; it may be a sensible strategy to place matches more likely to fail earlier in the file.

6.1 Property Expressions

The basic unit of most property expressions is the *predicate*. An exhaustive list of available predicates will be provided in a subsequent section of this manual. Aside from predicates, various familiar programming language logical connectives are provided:

```
P ::= B && B
P ::= B || B
P ::= !B

B ::= E == E
B ::= E != E
B ::= E <= E
B ::= E >= E
B ::= E < E
B ::= E > E

E ::= predicate(parameters)
E ::= integer literal
```

The essential execution model associated with these properties is that each time a new state is discovered and considered, the predicates are applied to *that state*. The simplest property we could define might be:

```
%property not_empty !empty();
```

Assuming we have some (pre-defined) predicate called `empty` that takes no arguments, then this property will be satisfied iff every state in the model is not empty.

It turns out that this is sufficient to write specifications that inspect the properties of individual states, but what if we want to write a specification that defines properties over states as they evolve? We have another mechanism for that:

```
%property growth size() >= $pred->size();
```

This property assumes that we have a pre-existing predicate called `size` that returns the "size" of a given term (spoiler alert: we do). The first application of the predicate `size` will evaluate to the size of the current state under consideration. The second instance of the predicate is prefixed with `$pred` (short for *predecessor*), which is a placeholder for "the state from which the current state was created by a step of reaction", so in this instance `size` is being applied to the predecessor state, not the current state. Finally, taking the greater-than-or-equal-to connective `>=`, this is a property stating that this model must never shrink through a step of reaction. If the checker can find any two consecutive states in the transition system where a state is smaller than the one from which it is derived, this will constitute a violation of the '`growth`' property.

6.2 Pre-defined Scopes

The full list of scopes:

- `$pred` The predecessor to the current state, such that there exists some step of reaction from `$pred` to the current state.
- `$this` The current state.
- `$succ` The set of successor states, i.e., those reachable by a step of reaction from the current state.
- `$terminal` The predicate in question is only applied to states marked `terminal`, that is, they do not lead to any further states and there are no reactions that can be applied to them. `$terminal->p()` will return `true` when applied to non-terminal states, or `p()` if applied to a terminal state.
- ... More to come? Suggest more useful scopes!

It's important to note that scopes like `$succ` and `pred` will not work when in local checking mode!

6.3 Pre-defined Predicates

6.3.1 `size()` Predicate

This predicate returns a count of the number of place-graph nodes in the current agent. For example:

```
a.a.a.nil --> 3
a.nil | b.nil --> 2
nil | nil | nil --> 0
nil --> 0
```

This can be used to construct properties such as:

```
%property growth $pred->size() <= size();
```

6.3.2 `matches(`*t*`)` Predicate

This checks whether a given redex *t* matches anywhere within the current agent, subject to the usual active contexts restriction. For example:

```
%active a : 0;
%active b : 0;
%active c : 0;

a.b | a.c;

a.$1 | $0 -> c.$1 | $0;
```

```
%property cc !matches(c.c);

%check
```

6.3.3 `empty()` Predicate

This is equivalent to the property `size() == 0`.

6.3.4 `terminal()` Predicate

This checks the property of the current node in the transition system and returns `false` if the node has outgoing edges (i.e. there are further agents reachable by the application of some reaction rule to this agent), or `true` otherwise. See '`doc/examples/dining.bgm`' for an example use of this predicate to define deadlock-freedom.

7 Checking

Having defined your reaction rules, model and properties, the final line in your file should be `%check`. This signals to `bigmc` that checking should actually begin.

For example, we could check some '`diverge_prop.bgm`' (which is in the '`doc/example`' directory of your BigMC distribution):

```
%active a : 0;
%active b : 0;
%active c : 0;

a.b;

b -> a.b;
a.a.b -> a;
a.$0 -> c.$0;

%property growth size() >= $pred->size();

%check;
```

We invoke `bigmc` using a command such as:

```
$ bigmc diverge_prop.bgm
```

This will result in the following backtrace:

```
*** Found violation of property: growth
*** growth: size >= $pred->size()
#0  a.nil   <- *** VIOLATION ***
 >> a.a.b.nil -> a.nil

#1  a.a.b.nil
 >> b.nil -> a.b.nil

#2  a.b.nil
 >> (root)
mc::step(): Counter-example found.
```

We interpret the backtrace as showing us a state that violated the property `growth`. States are displayed from "newest" to "oldest", interleaved with the reaction rule that was applied to reach that state from the previous one.

8 Example

This example can be found in '`doc/examples/airport2.bgm`' in the BigMC distribution.

```
############################################################################
# Airport Example #2
# Gian Perrone, August 2011
#
# This is a very crude model of an airport with two passengers.
# Passengers travel to their flights through a series of steps,
# and enter a gate based upon their associated record in the
# passenger database.
######################## Definitions Section ##########################
# We distinguish landside and airside

%active Landside : 0;
%active Airside : 0;
%active Gates : 0;

# We distinguish zone types
# The two links are [identifier, exit]

%active Zone : 2;

# A gate is simply linked to a flight
%active Gate : 1;

# A passenger is linked to a flight
%passive Pax : 1;

# The database store
%active DB : 0;
%active PaxRecord : 2; # Links [passenger,gate]

# Names that represent zones
%name CheckIn;
%name Security;
%name DutyFree;
%name GateLounge;

# A gate status at the departure lounge
%name Boarding;

# Some flights
%name SK100;
%name SK101;
%name SK102;
```

```
# Two example passengers
%name Bob;
%name Joe;

######################### Reaction Rules ##############################

# Rule that allow passengers to move through the airport
Zone[w,x].(Pax[y] | $0) || Zone[x,z].$1 ->
Zone[w,x].$0 || Zone[x,z].(Pax[y] | $1);


# A passenger linked to a PaxRecord can proceed to the gate
Zone[m,Boarding].(Pax[y] | $0) || Gate[x].$1 || PaxRecord[y,x] | $2 ->
Zone[m,Boarding].$0 || Gate[x].(Pax[y] | $1) || PaxRecord[y,x] | $2;

# A passenger magically disappears once they board the aircraft
Gate[x].(Pax[y] | $0) -> Gate[x].$0;

######################## Airport Model  ###############################

Landside.(
Zone[-,CheckIn].(Pax[Bob] | Pax[Joe]) |
Zone[CheckIn,Security]
) |
Airside.(
Zone[Security,DutyFree] |
Zone[DutyFree,GateLounge] |
Zone[GateLounge,Boarding]
) |
Gates.(
Gate[SK100] |
Gate[SK102] |
Gate[SK101]
) | DB.(
PaxRecord[Joe,SK100] |
PaxRecord[Bob,SK101]
);

%check
```