Mace ModelChecker How-To

Meg Walraed-Sullivan, Chip Killian

October 6, 2009

1 Overview

Model checking involves several files, including multiple layers of services/applications. The service to be checked sits in this stack of layers, above whichever services it uses (such as transport layer services), and below the *SimApplication* service, the service which emulates an application in order to test the checked service. Above the *SimApplication* is the modelchecker itself. One other file is necessary, and is usually called something like *ServiceTests.cc*. The modelchecker uses this file to build the hierarchy of services and to load its tests, once for each execution of the service to be tested. The following will go through an example using the `BrokenTree` service. Figure 1 depicts the relationships between the various files used to modelcheck a service.

2 Files

This section contains an overview of the files necessary for the `BrokenTree` example.

2.1 BrokenTree Service

The `BrokenTree` service is based on a protocol used to reorganize a tree periodically in such a way that bandwidth is distributed according to a particular scheme. An assumed invariant of the correct version of this protocol is that no loops are ever formed; however, the `BrokenTree` protocol allows for such loops to be formed, i.e. it is broken. The code for `BrokenTree` can be found in the file *BrokenTree.mac*.

Note that the `BrokenTree` service has 3 states, `init, joining, and joined`. Each node keeps track of its parent and children, and uses this info to suggest reorganizations to its children, based on their parents, grandparents, and children. A node joins the tree by sending a `Join` message to its parent. The parent replies with a `JoinReply` message. A timer (`reorg`) is set at the time of this request so that the node can retry its request if necessary. The root

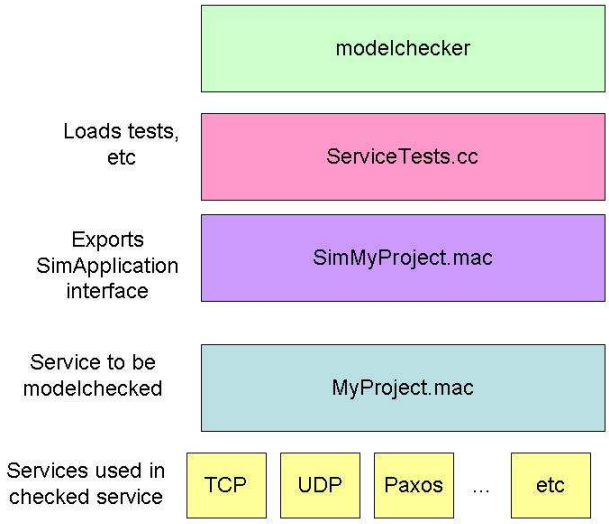

Figure 1: Service/Application Stack

overloads this timer in order to decide when to start a reorganization. When it decides to do so, it sends a message to its children, who recursively continue the reorganization process. To work through this example:

- If it does not already exist, create a directory called *BrokenTree* under *[Path-to-Mace-Source]/services*.
- Copy *BrokenTree.mac* into this directory.
- Copy *CMakeLists.txt.BrokenTree* into this directory and remove *.BrokenTree* from the file name.
- Open *[Path-to-Mace-Source]/services/CMakeLists.txt* and add `BrokenTree` to the line beginning with `SET(SERVICE_DIRS`, if not already present. This tells Mace to build the `BrokenTree` service.

2.2 SimTreeApp Service

The `SimTreeApp` service is the service used to test the `BrokenTree` service. Code for this service can be found in the file *SimTreeApp.mac*. The code is explained below:

```
service SimTreeApp;

provides SimApplication;
trace=med;
```

The name of this service is `SimTreeApp` and it provides the `SimApplication` interface (which is characterized by the implementation of the `eventsWaiting` and `simulateEvent` downcalls).

```
constants {
  int INIT = 0;
  int JOINED = 1;
  int ROOT_ONLY = 0;
  int NEXT_NODE = 1;
}
```

The ROOT_ONLY and NEXT_NODE constants are used to make this application a generic tree simulator. They determine how a node joins the tree, by using the root or by using the next node as a "parent" via which to join.

```
constructor_parameters {
  int PEERSET_STYLE = ROOT_ONLY;
  int SHOULD_JOIN = 1;
  MaceKey me = MaceKey::null;
  int num_nodes = -1;
}
```

For the purpose of this example, nodes join via the root.

```
states {
  needJoin;
  done;
}
```

This application keeps track of two states. The `needJoin` state indicates that the current node needs to send a `Join` request. The `done` state indicates that the node has sent such a request. *Note that it does not indicate that such a request has been received by the parent or accepted.*

```
services {
  Tree tree_;
  Overlay treeo_ = tree_;
}
```

This service implements both the `Tree` and `Overlay` interfaces.

```
transitions {

  downcall (state == init) maceInit() {
    state = needJoin;
  }
```

When it begins, this service has not yet sent a `Join` request.

```
  downcall (state == done) eventsWaiting() {
    return false;
  }
```

The `eventsWaiting` downcall must be implemented in any service which implements the `SimApplication` interface. It returns either `true` or `false`, in order to let the modelchecker know whether more events need to be run on behalf of the application. If the service is in the `done` state, the node has sent a `Join` request. The sending of a `Join` message the only application event for this example, so the `eventsWaiting` downcall always returns false from this downcall.

```
downcall eventsWaiting() {
  return true;
}
```

In any other state besides `done`, the `eventsWaiting` downcall can return true, since possible such states include `init` and `needJoin`, both states which hold before the `Join` request is sent.

```
downcall (state == needJoin) simulateEvent() {
  state = done;
  NodeSet peers;
  if(PEERSET_STYLE == NEXT_NODE) {
    peers.insert(upcall_getMaceKey((upcall_getNodeNumber()+1) % upcall_getNodeCount()));
  } else {
    peers.insert(upcall_getMaceKey(0)); //BrokenTree and Overcast expect the root to be passed in.
  }
  downcall_joinOverlay(peers);
  return "joinOverlay(" + peers.toString() + ")";
}
```

Based on the type of join, when the modelchecker calls the `simulateEvent` downcall, the node attempts to join the tree and changes its state accordingly. A string indicating the "short verion" of what the event did is returned.

```
downcall (state == init) simulateEvent() {
  maceInit();
  return "maceInit()";
}
```

In the `init` state, when the modelchecker calls the `simulateEvent` downcall, the `maceInit` downcall is called for this service. It is interesting to note that for this type of service, the invoking application, i.e. the modelchecker, does not call `maceInit` itself, as is generally the case for Mace services.

To work through this example [Note that none of these steps should be necessary with a standard Mace repository]:

- If it does not already exist, create a directory called *SimApplication* under *[Path-to-Mace-Source]/services*.
- If one does not already exist, copy *SimTreeApp.mac* into this directory.
- If one does not already exist, copy *CMakeLists.txt.SimTreeApp* into this directory and remove *.SimTreeApp* from the file name.

- Open *[Path-to-Mace-Source]/services/CMakeLists.txt* and add `SimApplication` to the line beginning with `SET(SERVICE_DIRS`, if not already present. This tells Mace to build the `BrokenTree` service.

2.3 BrokenTreeTest

This file tells the modelchecker how to create and invoke each of these services. It implements the method `loadTest` which is called by the modelchecker on every execution of the service to be tested. The following code can be found in *BrokenTreeTest.cc*.

```
namespace macemc {

#ifdef UseBrokenTree
```

This is included so that the tests for multiple services can be combined into one file.

```
class BrokenTreeMCTest : public MCTest {
  public:
    const mace::string& getTestString() {
      const static mace::string s("BrokenTree");
      return s;
    }
```

This function is required for the modelchecker to determine which service it is testing. It must match the `CHECKED_SERVICE` parameter in the *params.default* file.

```
    void loadTest(TestPropertyList& propertiesToTest,
                  ServiceList& servicesToDelete,
                  NodeServiceClassList& servicesToPrint,
                  SimApplicationServiceClass** appNodes,
                  int num\_nodes) {
```

The modelchecker passes *BrokenTreeTest.cc* a variety of arguments to fill in, via this function. The `TestPropertyList` is of type `mace::vector<TestProperties*, mace::SoftState>`. `TestProperties` is a list of properties to be tested (as specified in *SimTreeApp.mac*, *BrokenTree.mac*, or as pre-defined in *[Path-to-Mace-Source]/application/modelchecker/Properties.h*) and `mace::SoftState` is an indicator that the properties are not to be serialized. The list of `servicesToDelete` is created to help the modelchecker know what services must be cleaned up after it is finished. Transport services do not need to be added to this list but wrappers do. For instance, the TCP service needs not be added to the list but the wrapper used to include the `forward` upcall does need to be added. The `servicesToPrint` list stores services to print when the modelchecker displays output. The `appNodes` argument stores an array of pointers to the the `SimTreeApp` Service, one for each node, and the `SimTreeNodes` map stores a mapping from node number to `SimTreeApp` service. The modelchecker passes the `appNodes` argument to this method and `appNodes` differs from the `SimTreeNodes` map in interface and use. For instance, `appNodes` has the ability to allow properties to be tested using the information it stores. The `num_nodes` variable stores the number of nodes in the system.

```
int base_port = params::get("MACE_PORT", 5377);
int queue_size = params::get("queue_size", 20);
```

The `base_port` variable is used to specify the port for Mace to use. However, this is not necessary for modelchecking, since everything is simulated locally. It is included here because it will be required when the TCP service is instantiated. The `queue_size` variable indicates the queue size to be used for TCP. Both of these arguments can be specified in the *params.default* file.

```
mace::map<int, BrokenTree_namespace::BrokenTreeService*, mace::SoftState> brokenTreeNodes;
mace::map<int, SimTreeApp_namespace::SimTreeAppService*, mace::SoftState> simTreeNodes;
```

These two maps store pointers from each node to a service, one set of pointers for the `BrokenTree` service, and one for the `SimTreeApp` service.

```
for(int i = 0; i < num_nodes; i++) {
```

Everything is initialized once for each node in the system.

```
  ServiceClassList list;
```

This list is used to keep track of all services created for this node. Later, these lists of services are added to the `servicesToPrint` list.

```
Sim::setCurrentNode(i);
MaceKey key = Sim::getCurrentMaceKey();
SimulatorTCPService* tcp = new SimulatorTCPService(queue_size, base_port, i);
RouteTransportWrapper_namespace::RouteTransportWrapperService* rtw
     = new RouteTransportWrapper_namespace::RouteTransportWrapperService(*tcp);
servicesToDelete.push_back(rtw);
list.push_back(rtw);
```

The TCP service is created with the queue size specified in the *params.default* file, and the wrapper service is created as well. The wrapper service is pushed onto the list of services to clean up after modelchecking, as well as onto the list of services created for this node.

```
BrokenTree_namespace::BrokenTreeService* tree
     = new BrokenTree_namespace::BrokenTreeService(*rtw, Sim::getMaceKey(0), num_nodes);
servicesToDelete.push_back(tree);
list.push_back(tree);
brokenTreeNodes[i] = tree;
```

The `BrokenTree` service is created and pushed onto the list of services to clean up after modelchecking, as well as onto the list of services created for this node. The `brokenTreeNodes` map is updated to point to the appropriate instance of the `BrokenTree` service for this node.

```
      SimTreeApp_namespace::SimTreeAppService* app
           = new SimTreeApp_namespace::SimTreeAppService(*tree,
                                                         *tree,
                                                         SimTreeApp_namespace::ROOT_ONLY,
                                                         0,
                                                         key,
                                                         num_nodes);
      servicesToDelete.push_back(app);
      list.push_back(app);
      servicesToPrint.push_back(list);
      simTreeNodes[i] = app;
      appNodes[i] = app;
    }
```

The `SimTreeApp` service is created with a pointer to the `BrokenTreeApp` service and pushed onto the list of services to clean up after modelchecking, as well as onto the list of services created for this node. The `simTreeNodes` map is updated to point to the appropriate instance of the `SimTreeApp` service for this node, as is the `appNodes` array. The list of services created for this node is now added to the `servicesToPrint` list, which is a list of vectors, one vector of services for each node.

```
    propertiesToTest.push_back(new
         SpecificTestProperties<BrokenTree_namespace::BrokenTreeService>(brokenTreeNodes));
    propertiesToTest.push_back(new
         SpecificTestProperties<SimTreeApp_namespace::SimTreeAppService>(simTreeNodes));
  }
```

The properties found in the two service files are pushed back on the list of properties for the modelchecker to verify.

```
    virtual ~BrokenTreeMCTest() {}
};

void addBrokenTree() __attribute__((constructor));
void addBrokenTree() {
  MCTest::addTest(new BrokenTreeMCTest());
}
}
```

This code is "gcc magic" to allow for static initialization of the above class. This lets the class be initialized by the compiler so that users of the modelchecker do not have to alter the modelchecker's source to call the correct `loadTest` method each time they test a new service.

To work through this example:

- Copy *BrokenTreeTest.cc* into *[Path-to-Mace-Source]/application/modelchecker*.
- Open *[Path-to-Mace-Source]/application/modelchecker/CMakeLists.txt* and add the line `SET(modelchecker_SRCS BrokenTreeTest.cc EXTERNAL_TEST_SRCS)`, replacing any similar such lines.

2.4 Params.default

The *params.default* file file specifies arguments which are necessary for the particular service to be tested (the usual use for a *params* file), as well as properties used to tell the modelchecker how to run tests of the service. The included sample file is explained below.

```
MACE_PRINT_HOSTNAME_ONLY = 1                    # Required for perl to properly print Macekeys
MACE_LOG_LEVEL = 1                              # This can be increased for more logging but should not be
                                                #    decreased.


USE_UDP_REORDER = 0                             # Whether to reorder UDP messages
USE_UDP_ERRORS = 0                              # Whether to allow UDP errors
USE_NET_ERRORS = 0                              # Whether to allow network (e.g. socket) errors
SIM_NODE_FAILURE = 0                            # Whether to allow for node failure


USE_BEST_FIRST = 0                              # Selects between the Search Random Util and Best First Util.
                                                #    Best First maintains state about the tree, so that when
                                                #    a particular prefix is encountered twice, it does not
                                                #    need to be repeated.  This speeds up execution by about
                                                #    5% but costs extra memory.



MACE_PORT=10201                                 # Port for mace to use, unimportant for modelchecking
max_num_steps=80000                             # This is the total length of an execution.  Approximates infinity
search_print_mask=0                             # "Granularity" of print, says how often (in number of steps) to
#search_print_mask=15                           #    print state about the current execution
#search_print_mask=255
#search_print_mask=4095
PRINT_SEARCH_PREFIX=0                           # When true, every path that reaches a live state is printed once t
                                                #    live state is reached.


USE_RANDOM_WALKS=1                              # Whether to use walks after tree search, necessary for
USE_GUSTO=1                                     #    liveness properties
                                                # Turns off bad things (failures, etc) at random walk part,
USE_STATE_HASHES=0                              #    adds weights to types of events (application, network, timer)
                                                # Helpful to turn on but may not work.  To test, run MC and search
                                                #    for "non printable" in output


NUM_RANDOM_PATHS=40                             # Number of paths to search for each path in "binary search" during
#MAX_PATHS=20                                   #    last nail search
#MAX_PATHS=200
#MAX_PATHS=50000
#MAX_PATHS=1000000


COMPARE_CMC = 0                                 # For comparison with another modelchecker
RUN_CMC_MEM = 0                                 # For comparison with another modelchecker


RUN_DIVERGENCE_MONITOR = 0                      # Used to detect loops in code (by detecting long-running transitio
divergence_assert = 1                           # Whether to assert on timeout of divergence monitor
divergence_timeout = 30                         # Timeout for divergence monitor


CHECKED_SERVICE=BrokenTree

                                                SEARCH_DEPTH=5
#BrokenTree
num_nodes=4                                     # Which service to run
queue_size=100000
```

```
# Number of nodes and all parameters required by tested service
# MC needs to know how many nodes to simulate        # Bound for the iterative depth first search
# Used for TCP
```

3 Properties

Properties for the service to be tested can be specified in either the service file for the service being tested, or in the application-emulating service. For this example, the properties are listed in *SimTreeApp.mac*.

3.1 Property Specification

The grammar specification for properties is located (in a Recursive Descent Perl format) in *perl5/Mace/Compiler/MaceGrammar.pm*. The top-level rule to look at is "properties", which uses GrandBExpression and others. The idea of the language is to use a latex-like language. In large part, a goal was to be able to copy/paste the property into a latex document, and have it generate the mathematical formula to be tested.

Some common operators and expressions include:

- equality: =
- inequality: $\backslash neq$
- Element-Set $\backslash in$, $\backslash notin$
- Set-Set: $\backslash subset$, $\backslash propersubset$, $\backslash eq$ (seq equality, not C++ equality)
- Numerical: = (C++ equality), `<, >`, $\backslash geq$, $\backslash leq$
- Logical: $\backslash implies$, $\backslash and$, $\backslash or$, $\backslash xor$, $\backslash iff$, $\backslash not$

All properties presently have to start with a quantification over the nodes (special variable $\backslash nodes$). The quantification options are (n can be any variable name):

- $\backslash forall$ n $\backslash in$ $\backslash nodes$:
- $\backslash exists$ n $\backslash in$ $\backslash nodes$:
- $\backslash for${OP}{COUNT} n $\backslash in$ $\backslash nodes$:, Where OP is `<, >,` $\backslash geq$, $\backslash leq$, or =, and COUNT is a number.

```
So \for{=}{4} n \in \nodes would be "for exactly 4 n in nodes".
```

You can use 'n' as a reference to a node, and use the dot operator to access its state variables, 'state', or a const transition/routine.

Any variable which is a MaceKey can be "promoted" to a node reference. That is, suppose your service had a MaceKey variable "p" where it stored the address of a peer.

Then you could write a property:

```
\forall n \in \nodes: n = n.p.p
```

which would say that for each node, n's peer is n (or n is the peer of its peer).

You can also compute the closure of a recursion, in a property like this:

```
\forall n \in \nodes: n.p(.p)* \eq \nodes
```

Which would be: The set containing n.p, and the closure formed by continually adding new nodes to the set by referencing each node's .p is the set of all nodes.

We use this property to check that successor pointers in Chord form a ring of all nodes. You can also take the cardinality of a set like this:

```
\forall n \in \nodes: |n.p(.p)*| = 4
```

Which would test that for each node n, the closure of n.p(.p)* has size exactly 4.

You can use multiple quantifications, and parentheses to designate order of operations. This gives you something like:

```
\forall n \in \nodes: (n.state = init \or (\exists m \in \nodes: m.p = n))
```

Which would say, for all nodes n, either n is in the init state, or there exists a node m such that m.p = n. The parentheses may not actually be needed here, but can provide clarity. Additionally, curly braces can be used instead of parentheses for nesting, as desired for clarity.

Each property should begin with a property name and a colon, contain a "grand boolean expression" (GrandBExpression in the grammar), and end with a semicolon.

3.2 Properties for `BrokenTree`

The code shown at the end of *SimTreeApp.mac* is detailed in the following.

```
properties {
  safety {
```

`Safety` properties must be true at the end of each step.

```
    //     joinedImpNotNullParent : \forall n \in \nodes :
                { n.downcall_isJoinedOverlay() \iff ( n.downcall_getParent(MaceKey::null) \neq \null ) };
```

This property specifies that every joined node has a parent, and that every node with a parent is joined.

```
    //     isRootEqSelfParent : \forall n \in \nodes :
                { n.downcall_isRoot(MaceKey::null) \iff ( n.downcall_getParent(MaceKey::null) = n ) };
```

This property specifies that the root must list itself as its parent and that any node claiming to be its own parent is the root.

```
    noLoops : \forall n \in \nodes :
                { \exists m \in n.(downcall_getParent(MaceKey::null))* :
                                    { m.downcall_isRoot(MaceKey::null) \or \not m.downcall_isJoinedOverlay() } };
}
```

This property specifies that there are no loops within the tree hierarchy.

```
liveness {
```

`Liveness` properties must eventually be reached.

```
    allJoined : \forall n \in \nodes : n.downcall_isJoinedOverlay();
```

Eventually, all nodes must have joined the overlay.

```
    oneRoot : \for{=}{1} n \in \nodes : n.downcall_isRoot(MaceKey::null);
```

Eventually, only one node may think itself the root.

```
    kidsMatch : \forall n \in \nodes :
            {
              (n.downcall_isRoot(MaceKey::null)
                \or n
                \in n.downcall_getParent(MaceKey::null).downcall_getChildren(MaceKey::null))
              \and
              (\forall m \in n.downcall_getChildren(MaceKey::null) : n = m.downcall_getParent(MaceKey::null))
            };
}
}
```

Eventually, it must be the case that every node is either the root or is one of its parent's children, and all of each node's children must believe that node to be their parent.

4 Building the ModelChecker

The modelchecker is built with Mace, so now that all files are moved to the proper place, simply build Mace as usual:

- Create a build directory at *[Path-to-Mace-Source]/builds*.
- Move to *[Path-to-Mace-Source]/builds*.
- Generate makefiles: `cmake ../`
- Build Mace: `make`

5 Running the ModelChecker

Finally, to run the modelchecker:

- Move to the directory in which the modelchecker was built: *[Path-to-Mace-Source]/builds/applicati*
- Create a test directory to store output and cd into this directory.
- Copy *Params.default* into the test directory.
- Run the modelchecker: `./modelchecker params.default`
- Alternatively, the modelchecker can be run with a perl script which can be copied into the build directory for the modelchecker (*[Path-to-Mace-Source]/builds/application/modelchecker*) from *[Path-to-Mace-Source]/application/modelchecker*. In this case, run with `../run-modelcheck` from within the test directory.

As it runs, the modelchecker will create several files. *error.log* and *error.graph* are useful for debugging, as are *live.log* and *live.graph*, if created.

6 Debugging with the ModelChecker

Use `mdb error.log` to step through and graphically view a trace. Once running mdb, typing h will display a list of commands, enabling you to move forward or back a step, view the event graph, etc. The following gives more detailed explanations for these commands: [TODO: this is not more detailed right now!]

- h: print help
- j: jump to step
- n: next step
- p: previous step
- P: display current step in pager

- l: list current step
- L: list step to external program state-dump.sh
- s: regex search
- d: diff states
- m: diff states across log files (non-interactive only)
- I: filter include selectors
- E: filter exclude selectors
- i: filter include
- e: filter exclude
- w: select which node to step
- t: disable/enable properties
- c: clear the screen
- x: search all steps
- g: generate event graph
- q: quit
- r: choose random number (interactive only)
- a: show possible next steps (interactive only)
- R: run until live (interactive only)
- B: branch modelchecker from current step (interactive only)
- P: print to file: path, path to current step, event graph, output log (interactive only)
- G: toggle gusto (interactive only)