Interfacing and Optimization of Synthesis for Modal Logic

M. Blom
INTERFACING AND OPTIMIZATION OF SYNTHESIS FOR MODAL LOGIC

FIRST INTERNSHIP

M. Blom - 0590314

Report Number: 420759

Supervisors:
Dr. ir. M. A. Reniers
Drs. A. C. van Hulst

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF MECHANICAL ENGINEERING
SYSTEMS ENGINEERING GROUP

May 26, 2014
Recently, new developments have been made in the synthesis for a modal logic [1], including invariant and reachability properties. A very basic tool was created in C to perform this synthesis, using a text-based automaton specification.

The purpose of this tool is to restrict the behaviour of an automaton by synthesizing it with a control requirement declared as a modal formula, while maintaining maximally permissive behaviour of the automaton.

The goal of the first part of this project is to make this tool more accessible for a wider user group, including students. This is done by integrating the synthesis tool in the Supremica toolset. By doing this, automata can be specified using the Supremica graphical user interface instead of a text-based description and the modal formula or requirement can be synthesized directly from the user interface instead of the command line. Also, it is possible to synchronize multiple automata with Supremica, so that a system can be specified as a network of automata, and it is possible to use variables when specifying input automata.

The second part of this project is aimed at improving the synthesis algorithm with respect to the checking of reachability properties. Since the algorithm needs to recheck every reachability (sub)formula after the removal of a state or transition there are improvements to be made.

This report starts with the integration of the synthesis tool in Supremica in chapter 2. The changes made to the synthesis tool and Supremica will be discussed and the structure of the tool will be explained. Chapter 3 will treat the changes made to the reachability function of the synthesis tool. Various different algorithms and approaches are discussed. In chapter 4 the methods of comparing the different algorithms are discussed as well as the random model generator and the formula generator. Also, a few small Linux shell scripts used to run the tests are explained. Chapter 5 discusses the results from these tests and suggests different subjects for future work.
Chapter 2

Synthesis tool integration

2.1 Supremica as a GUI

Since writing an entirely new GUI for the synthesis tool is too much work and
not worth the effort, the decision was made to implement the tool in an already
existing toolset. The synthesis tool uses the following characteristics in the
representation of a system:

- States, including one initial state
- Transitions between states with events assigned to them
- Labels assigned to states

The synthesis tool uses labels assigned to states in order to synthesize modal
formulas. These labels represent properties that hold in one or more states.

An example is given in figure 2.1. The traffic can be working (ok) or not
working (broken). States x, y and z represent the traffic light when it is in
normal operating mode. In these states the traffic light is both ok and has a
color. With each event t a transition is made to a next state. If the traffic light
breaks down the transition e is made to state p. In this state the properties
broken and orange hold for the traffic light. The traffic light will then start
blinking, but the property broken will hold whether the orange light will be on
or off.

Aside from the characteristics mentioned above, certain features are taken
into account when selecting a toolset:

- Does the toolset have a Graphical User Interface (GUI)?
- Does the toolset support state labels?
- Can the toolset synchronize multiple automata?
- Is the source code of the toolset available?

When looking into several different toolsets the criteria mentioned above
had to be taken into account. With these criteria in mind, there were several
options for tool implementation:
Figure 2.1: Example of a model representation of a traffic light that can break down

- CIF [2]
- UPPAAL [3, 4]
- Supremica [5, 6]

Both CIF and Supremica have the ability to synchronize multiple automata into one, which is an advantage. However, at this point CIF 3.0 does not have a Graphical User Interface (GUI), which makes it less accessible for students who are new to modeling. UPPAAL already handles model checking with formulas and has a GUI, but lacks the ability to synchronize multiple automata into one. Also, the UPPAAL source code is not publicly available at the time of writing.

None of the tools mentioned above natively support the use of labels assigned to states. They do, however, use (unique) state names assigned to states. Since for the synthesis tool state names are irrelevant, the choice was made to use Supremica as a GUI, and to interpret the state names as labels for the synthesis tool. In the synthesis tool states will simply be numbered instead of using the names assigned by Supremica.

In Supremica, when multiple automata are synchronized into one, the state names from the original automata are concatenated with a period in between (figure 2.2). The synthesis tool appends all those concatenated state names as labels to the corresponding state so that multiple properties can hold in a single state.

When creating an automaton in Supremica with a state that has more than one label, simply separate the labels with a colon in the state name. The synthesis tool will identify the colon as a label separation sign.

The only problem that could arise when modelling an automaton in Supremica is when two or more states have an identical single label or label concatenation. Since Supremica only supports automata with unique state names, there can not exist two states with for example the name ok. To circumvent this, one could simply add another (meaningless and/or random) label to the state, resulting in for example the two state names ok:1 and ok:2.
2.2 Original synthesis tool automaton specification

The original synthesis tool created by Allan van Hulst requires a specific automaton specification format, which is read from a text file. The tool is ran from the terminal and the output is printed to the screen. An example of such a specification can be seen in figure 2.3.

The first line of the specification states that state $s$ is the initial state. Lines 2, 3 and 4 denote transitions in the format `<start state>;<event>;<target state>`. Lines 5 and 6 assign the label $p$ to state $y$ and the label $q$ to state $x$ respectively in the format `<state>:<label>`. These lines do not have to be in any specific order, but are organized in this example for clarity.
2.3 Tool adaptation

There are three different options for tool integration with Supremica:

1. Write the synthesis code in Java, making it an integral part of Supremica.
2. Export the system specification to the tool-specific format and call the synthesis program from Java.
3. Export the automaton specification to XML (an existing functionality in Supremica), adapt the synthesis tool to accept the XML automaton specification format and call the synthesis program from Java.

The first option is slower than the other two with regard to the actual synthesis since Java is an interpreted language and C has direct access to the system resources. Another disadvantage of the first option is that further development of the synthesis tool would always have to be within Supremica. Running the tool from the command line would be impossible. With large models and future development for the tool in mind, this option was discarded. After careful consideration the decision was made to adapt the synthesis tool to accept XML model specifications for the following reasons:

- The synthesis tool is a relatively small program written in C, so it is easier to fully understand and adapt the code.
- It is easier to call an external program from Java than it is to write a new export and import function in Java.
- There exists an XML C parser and toolkit called LibXML2 which makes parsing and writing an XML file in C relatively easy.

2.4 Synthesis tool structure

In order to read and write the XML automaton specification files of Supremica the synthesis tool was augmented with several functions. The original program structure can be seen in figure 2.4, while the added or altered functions can be seen in red in figure 2.5. A description of the original functions and the added or altered functions can be found below.

The function $Main$ function checks if the program is run with the correct number of parameters. If this is not the case, help on the correct usage of the tool is outputted to the console. If the program is called with the correct number of parameters the process of parsing the automaton specification file and formula is started.

The $Read$ function of the original synthesis tool was specifically designed to parse the automaton specification format and could not parse the XML specification format of Supremica. The function was therefore replaced by new functions.

The function $parseFile$ opens the XML file exported by Supremica and checks for several possible errors:
• The file was not read successfully.
• The file is empty.
• The file is not a Supremica automaton specification file because it does not contain the XML root node Automata.
• The XML node Automata does not contain the XML child node Automaton.

Any possible errors are written to the Supremica console. If the file contains no errors the function parseModel is called.

The function parseModel iterates over all the children of the automaton to search for the nodes events, states and transitions. When one of these nodes is found one of the corresponding functions parseEvents, parseStates or parseTransitions is called.

The functions parseEvents, parseStates and parseTransitions iterate over all the children of the respective node and when a node is found the function addcons is called which adds a construct (initial state, label or transition) to the internal model.

When all components have been added the function synth is called and the actual synthesis is done via several other functions which are not discussed here.

The original synthesis tool was run from the command line, and therefore printed information to the screen. This part of the functionality is no longer needed for the synthesis and is turned off by default, but left in to help with debugging. To enable debugging, change the variable DEBUG in the C source code to 1. The parts of the function that print anything to the screen can be removed from the C code when no more augmentation or development of the code is needed.

After the synthesis, it is printed to the Supremica console whether or not the synthesis was successful. Next, the basis of the output XML structure is created in the modified part of the synth function. From there, the functions addEventsXML addStatesXML and addTransXML are called to iterate over all the respective components of the result structure and add them to the XML output structure. Finally, the XML file is written to disk with the xmlSaveFormatFileEnc command from the libXML2 library.
2.5 Supremica adaptation

In order to be able to call the synthesis program from the Supremica GUI several Supremica source files were adapted and the function `AnalyzerReachabilityAction` was added. The new function was added to the list of actions in the src/org.supremica.gui.ide.actions/Actions.java file, and buttons to call the function were added in the src/org.supremica.gui.ide/AnalyzerPopupMenu.java file and the src/org.supremica.gui.ide/IDEMenuBar.java file. This way, the synthesis program can be called easily from within Supremica (figure 2.6).

When the option for synthesis of a modal formula is chosen, the function `AnalyzerReachabilityAction` checks if a single automaton is selected (figure 2.7). If this is the case, a dialog box is created where the user can enter the modal formula.

After entering the formula the function `Tester.parser` (written by Allan van Hulst) is called to verify the syntactical correctness of the entered formula. If the formula is not correct the function attempts to return useful information with respect to the incorrectness of the entered formula.

When all checks are complete the automaton is exported to an XML file with a randomized filename in the users temp folder. This is done by calling the Supremica function `AutomataToXML`. Finally the synthesis tool is executed and the output from the tool is read. If the synthesis is completed successfully the resulting file is opened into a new module in Supremica. The new Supremica module is given a name according to the following specification: `SMF(< synthesized automaton >,< synthesized formula >)`. If the syn-
Figure 2.5: Program structure of the modified synthesis tool. The functions in red are altered or newly added.
Figure 2.6: The option to run the synthesis tool from within Supremica.
Figure 2.7: Interaction between the user, Supremica and the synthesis tool.
thesis is not completed successfully the message *Synthesis was not successful* is printed to the console.

Figures 2.8 through 2.14 show an example of the synthesis of a modal formula in Supremica. This example shows two traffic lights which are modelled and synchronized in Supremica and then synthesized with a modal formula which ensures that always one of the traffic lights is red.

Figure 2.8: Two traffic lights are modelled in Supremica with each three states and three different transitions.
Figure 2.9: In the Analyzer tab, both automata are selected and the option *Synchronize* is chosen. The options for synchronization are left to default.
Figure 2.10: The resulting synchronized automaton. At this point, the state where both traffic lights are green exists and is reachable.
Figure 2.11: With the synchronized automaton selected, the option *Synthesis of modal formula* is chosen.

Figure 2.12: The formula is entered. The synthesis of this formula ensures that always one of the traffic lights is red.
Figure 2.13: The synthesis output message.

Figure 2.14: The resulting automaton. Either one of the traffic lights can do a complete cycle as long as the other is red.
Chapter 3

Synthesis algorithm optimization

3.1 Reachability

The synthesis tool differs from other tools and algorithms that work with modal logics in the aspect that during every iteration it is possible that states or transitions are removed [8]. The synthesis tool algorithm uses several functions for different modal logic formulas. The focus for the second part of this project is the algorithm used to synthesize the reachability formula $\Diamond p$. This formula holds if, in the current state, it is possible to reach one or more states where property $p$ holds.

3.2 Synthesis tool data structure

In order to discuss how certain algorithms are implemented in the synthesis tool, first the data structure used in the synthesis tool must be considered.

The model data structure (figure 3.1) consists of 5 main elements:

1. An initial state.
2. The number of labels in the model.
3. The number of transitions in the model.
4. The list of labels in the model.
5. The list of transitions in the model.

Each label data element has two properties: a state name and a state label. If a state in the model has more than one label, separate label data elements exist. These label data elements are not organised or sorted. If one would want to know what labels a certain state has, the whole list needs to be iterated.

Each transition data element has three properties: a start state, an event and a target state. As with the label data elements, these transition data elements...
are not sorted. If one would want to know what incoming or outgoing transitions a certain state has, the whole list needs to be iterated.

### 3.3 Original algorithm

The original synthesis tool uses a forward depth-first algorithm as shown in 3.4.1 to check reachability formulas. With each iteration of the synthesis function states or transitions might be removed from the model. The original tool rechecks all reachability formulas each iteration, regardless of whether any relevant transition was removed.

### 3.4 Algorithm improvements

#### 3.4.1 Backward reachability versus forward reachability

While most model checking software (NuSMV [7], Uppaal [3, 4]) use backward reachability algorithms, there are no conclusive results that show that this is actually faster than forward reachability. The paper written by Iwashita, Nakata and Hirose [9] actually suggests the opposite. To see which algorithm performs better in this case both forward and backward algorithms have been implemented in different versions of the synthesis tool.

The algorithm used for backward reachability in model checking software is taken from the book *Principles of Model Checking* [10, p350-351] and the steps are briefly explained here for the formula $\diamond p$:

1. Collect all states where property $p$ holds and mark them
2. Mark all states that have a transition to one of the marked states

3. Repeat until no new states are added

4. Now all states that can reach $p$ are marked and the formula $\Diamond p$ can easily be checked for any state

While this might be a suitable solution for model checking software, it is not suitable for implementation in the synthesis tool due to the model altering during iterations.

The forward reachability function that was used in the synthesis tool takes start state $S$ and property $p$ as input arguments. Then the function is called recursively for all states $t$ that are reached by a transition from $S$. The function returns true if property $p$ holds in the target state.

```java
function check_reachability(S, p) {
    if (S has property p) {
        return true;
    }
    if (S has been previously checked) {
        return false;
    }
    add S to list of checked states
    for (all states $t$ with transition from $S$) {
        if (check_reachability(t, p)) {
            return true;
        }
    }
    return false;
}
```

The backward reachability function that is implemented in the synthesis tool takes start state $S$ and target state $t$ as input arguments, where $t$ is a state where the reachability property $p$ holds. It then recursively checks all states $s$ that have a transition to $t$.

Due to the data structure of the synthesis tool, in order to find a target state $t$ the algorithm must iterate over all existing states to find a state where property $p$ holds. If such a state is found but the start state $S$ can not be reached, the reachability function is started again for the next state $t$ where $p$ holds.
function check_reachability(S, t)
{
    if (S == t) {
        return true;
    }
    if (t has been previously checked) {
        return false;
    }
    add t to list of checked states
    for (all states s with transition to t) {
        if (check_reachability(S, s)) {
            return true;
        }
    }
    return false;
}

Both the forward and the backward method iterate depth first over all transitions. It can not conclusively be said which is faster in theory due to the fact that these methods are highly model dependent. However, as the forward algorithm turned out to be much faster in this case (see section 5.1), this was used in the final version of the synthesis tool.

3.4.2 Path memory for reachability computation

Instead of rechecking each reachability formula after the removal of a transition, the function is altered to keep track of all the transitions that are used to reach a property. For each property, once a path has been found it is saved to memory and the algorithm stops. That way, if any transition is removed (simply set to NULL), the function can check if all transitions used in a reachability path still exist. If no relevant transitions are removed, there is no need to recheck the formula. However, if a transition that was used in the path was removed, the function will try to find a new path.

An example is given in figure 3.2 where the formula $\Diamond p \land \Box [b] p$ is synthesized (without maximality for simplicity). The first part of the formula $\Diamond p$ (property $p$ must be reachable) is checked and the path (transition 1 from $s$ to $x$ and transition 2 from $x$ to $y$) is saved to memory. Then, due to state $z$ failing to satisfy the formula $\Box [b] p$ (always, after a transition $b$ property $p$ must hold) transition 3 from $x$ to $z$ is removed. After removal of transition 3 it is checked if the formula $\Diamond p$ is still satisfied. Because both transition 1 and 2 still exist, reachability does not have to be checked again.

The algorithm implemented in the new synthesis tool is stated below. Each reachability (sub)formula that must be checked has a start state $S$, a property $p$ and a collection $T[]$ of $n$ transitions that are relevant to the verification of the formula. This collection $T[]$ of transitions is a list of all transitions that, starting in state $S$, are needed to be able to reach a state where property $p$ holds.
Figure 3.2: Property \( p \) is still reachable despite transition removal

In this function it is first checked if the reachability formula has been checked before. If this is not the case, the reachability check will be performed. If the formula has been checked before, it is checked whether all relevant transitions still exist. If this is not the case, the reachability check will be performed.

The \texttt{check\_reachability} function is adapted to add transitions to the \texttt{reach\_prop} (lines 8 and 9).
function check_reachability(S,p)
{
    if (S has property p) {
        return true;
    }
    for (all states t connected to S) {
        if (check_reachability(t,p)) {
            add transition T from S to t to reach_prop.T[];
            reach_prop.n := reach_prop.n + 1;
            return true;
        }
    }
    return false;
}

Line 8 of the check_reachability function states that if the current transition is part of the path that satisfies the reachability formula, it is added to the list of n used transitions. The number of transitions relevant to this reachability formula is then increased by one on line 9. Of course multiple paths could exist, but only a single path is saved to memory.
Chapter 4

Synthesis execution comparison

In order to compare the performance of the different reachability functions in the synthesis tool, a model generator and a formula generator were written to generate random models and formulas. The specifics of these generators are discussed in this chapter. Also, the various Linux shell scripts used to run these generators and the synthesis tool are discussed as well as the various versions of the synthesis tool and the method of testing.

4.1 Model generator

The model generator is called from the command line with 7 arguments:

1. Number of models: the number of models that are generated. Each model will be named random_model_X.xml, where X is a number starting from 0.

2. Depth: The maximum depth of the model. The generator starts by creating transitions from the initial state. These transitions point to target states. Each of these target states can have transitions to other states. Each iteration, the target states are assigned a number one higher than their parent state. When this number has reached the maximum depth, no transitions will be created starting in that state.

3. Breadth: The maximum outdegree of the model, the maximum number of transitions generated originating in one state per iteration.

4. Number of properties: the number of different possible properties that are used in the model. If this number is 2 for example, only properties a and b will be used.
5. Number of events: the number of different possible events that are used in the model.

6. Maximum number of states: If the maximum number of states is high compared to the depth $\times$ breadth of the model, target states for generated transitions will most likely be new states. If this number is low, the probability of transitions to existing states (loops) will be higher.

7. Label percentage: The probability as percentage that a generated state is assigned a label. For every transition generated to a state, the chance that that target state receives a label is set by the label percentage parameter. Since a transition may lead to an already existing state it is possible for states to have more than one label. These labels may be the same as a previously assigned label. The synthesis tool will filter duplicate labels when parsing the XML model specification file.

### 4.2 Formula generator

A formula generator program was written by Allan van Hulst and adapted in order to have a higher chance of generating formulas that test the reachability function of the synthesis tool. The formula generator is called from the command line with 4 arguments:

1. Salt: a random number to ensure that every time the program is called a new random formula is generated.

2. Depth: the maximum depth of the formula. A generated formula can be $true$, $false$, a property $p$, $\neg p$ or $\varphi$, but can also be an operator that requires one or two subformulas (for example $\Box$, $\land$ or $\lor$). Each of these subformulas are assigned a number one higher than their operator. If that number has reached the maximum depth, the subformula will not be an operator that requires another subformula.

3. Number of events: the number of different possible events that are used in the formula.

4. Number of properties: the number of different possible properties that are used in the formula.

The maximum number of events and/or properties is 26 at this time.

In order to generate formulas that give meaningful results when synthesized, the probabilities for the different components of a (sub)formula are not all the same. For example, at maximum depth the probability of generating $\varphi$ is $\frac{1}{3}$, while the probability of generating $true$ or $false$ is $\frac{1}{15}$. This is done because the only difference in algorithm is the reachability function.
4.3 Linux shell scripts

4.3.1 Creator shell script

This shell script calls the formula generator and model generator programs, and writes all generated formulas to the file `formulas.list`. Generated models are placed in a subfolder called `models`.

4.3.2 Do-Synth shell script

This shell script calls the synthesis tool for every combination of random model and formula that have been generated by the Creator shell script.

The synthesis tool is still under development at the time of writing. Early testing showed that there were still a few bugs present in the synthesis code that could cause memory leaks and result in an infinite synthesis duration or crashing servers. These bugs only showed when certain combinations of random formulas and models were synthesized. Therefore, every synthesis was given a timeout of 10 seconds. The timeout command is not needed on every machine, some Linux distributions abort the process by itself when memory leaks occur and the maximum amount of memory is exceeded.

4.4 Execution timing

The execution times of the synthesis tool can be acquired with the Linux shell command `time`, but the resulting synthesis durations would include I/O operations loading (and writing) the model. To circumvent this, the synthesis tool was edited to include a timer and to output the duration of the actual synthesis in milliseconds to the command line interface.

4.5 Synthesis tool modification

In order to test the different reachability functions, 4 different versions of the synthesis tool were compiled. These versions were the result of the different combinations of forward and backward reachability and with or without reachability path memory. These versions will be compared on execution time.

4.6 Testing

The tests were run on a Linux server with two Intel(R) Xeon(R) E5-2630 processors and 128GB RAM. The two processors have a total of 24 cores running at 2.30GHz. The operating system installed is Debian GNU/Linux 7.4. Every version of the synthesis tool was tested with the same combinations of 50 formulas and 100 models. A total of 4 tests with each 5000 combinations were run for each version of the synthesis tool.
4.7 Results

Small tests (500 combinations of models and formulas) comparing backward and forward reachability functions in the synthesis tool showed that the backward reachability function (figure 4.1) is much slower than the forward reachability function (figure 4.2). The backward reachability function had an average of 1.6 seconds per synthesis, while the synthesis tool using the forward reachability function had an average of 0.21 seconds per synthesis, making it about 8 times faster. It was then concluded that it was not interesting to run large tests, since the backward reachability function is not a viable option in this case.

When comparing the algorithms with and without path memory the results are a lot closer. As you can see in the box plot in figure 4.2 execution times are often very small. This is due to the randomness of the combinations of models and formulas. Very often it easy to check if a formula holds or not when for instance the formula states that property $a$ must be reachable and the initial state of the system has property $a$. 

Figure 4.1: Boxplot of the synthesis execution times with backward reachability function.
Figure 4.2: Boxplot of the synthesis execution times with forward reachability function.
Chapter 5

Conclusions

5.1 Backward reachability versus forward reachability

When comparing the results for synthesis with forward and backward reachability functions it was immediately clear that backward reachability is slower than forward reachability. Synthesis times are on average 8 times larger when a backward reachability function is used. This is probably due to the data structure used in the synthesis tool as mentioned in section 3.4.1.

5.2 Path memory for checked reachability

When comparing results for synthesis with and without reachability path memory, the reachability function with memory is on average 25% faster than without. Note that not all tested combinations of formulas and models resulted in the removal of transitions during the synthesis. Combinations of models and formulas that always cause a lot of transition removal during synthesis iterations could yield even better results.

5.3 Future work

In order to properly compare backward and forward reachability functions it is necessary to change the data structure of the synthesis tool so that it benefits backward reachability. Then, the combinations of data structures and functions can be analysed and proper conclusions can be drawn.

An interesting phenomenon that occurred during testing of the different functions was that formulas with the same properties, but in a different order (for example $p \land [a]q$ and $[a]q \land p$) had different synthesis times. Research into what causes this could result in a more optimized synthesis.
Due to lack of time it was not possible to compare the memory usage of the synthesis tool with different reachability functions implemented. It would be interesting to see the (possible) tradeoff between cpu time and memory usage.

Also there was not enough time to play around with the parameters for model and formula generation. Perhaps certain parameters could yield larger differences in results between the variations of the algorithm.
Bibliography


