Lecture Notes Program Verification

I.S.W.B. Prasetya
Dept. of Information and Computing Sciences
Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands

email: `wishnu@cs.uu.nl`
URL: `www.cs.uu.nl\~wishnu.html`

Contents

Chapter 1

Testing

1.1 Introduction

Let us abstractly view a program P as offering a set of *operations*. In Java you would call them methods. Each operation can take parameters, and when it is called it will trigger some execution of the program, hence moving it from one state to another. While it does so, the program may produce various responses, e.g. in the form of a return value, returned by the operation, or in the form of observable events.

These operations are assumed to be the only interface we have to interact with P. Therefore we can test P, by testing the operations. This is done by exposing P to one or more *test suites*. A test suite is just a fancy term people use to mean a set of *test cases*. A test case is a procedure describing:

1. how to drive P through a sequence of calls to P's operations.

2. how to collect P's responses, and query P's states.

3. how to check if those responses and state-queries satisfy certain expectations.

For the obvious reason we of course prefer to have programmatic test cases, also called *test scripts*. Except in this introduction part, in the rest of this chapter we will only discuss this kind of test cases.

In testing jargon, the target program that we test is often called *System Under the Test* or SUT.

In reality, there are various kinds of programs, and therefore also various kinds of SUTs. Some program may just have a single operation, e.g. if it is a C function, or a Haskell function. On the other hand, a Java class typically offers multiple methods as operations. A (purely) functional program would have no state, and its only response is its return value. On the other hand, a Java applet may produce lots events as reponses to a single click on its 'OK' button. Some program may require its operations to be called in some strict order, e.g. if it is a payment protocol. On the other hand, a Java's Collection class allows you to invoke its methods in any order.

Those differences do influence the choice of your testing approach. But the view that I gave above should be generic enough as a common base for a general discussion on testing.

As said above, a test case checks if SUT's states and responses satisfy certain expectations. The straight forward way to do this is by checking them against P's specification. However, this requires that we have the specification in a *machine-chekcable* form. In practice people do not usually have that kind of specifications.

Therefore, in practice 'expectations' are often programmed as concrete values, which to be compared with those responses and states. These 'concrete expectations' have to be manually calculated, which is obviouly is very labour intensive, and fragile against changes.

Adopting formal specification may seemingly solve this problem. But on the other hand, you would then need to make sure that your specifications are complete. For many programs, this would be very hard, and therefore expensive.

The pragmatic solution is probably somewhere in between. We could use *partial* specifications to complement traditional test cases with concrete expectations. So that in case we are having problems in calculating concrete expectations, we still have the partial specifications to fall back.

Testers often use the term *oracle*, which is an english word that can be freely translated to 'source of wisdom'. Testers use it to mean:

> A source to determine expected results to compare with the actual result of the software under test. An oracle may be a requirement document, a user manual, or an individual's specialized knowledge, but should not be the code.

A specification is thus an instance of oracle.

Testability

If the operations of an SUT can be easily and systematically approached from a test case (e.g. we can simply call them, as if they are Java methods), then we say that the SUT is *testable*. Otherwise, then it is less testable, which would make it hard to automate its testing. For example, a Java class is highly testable. A Java class that requires a framework (e.g. a J2EE's enterprise bean, that would require most of J2EE machineries to execute) is less testable. GUI applications tend to be very low in testability, because to test them requires us to click and drag on screens. It is difficult to relate points in the screen, with functionalities of the target program. Furthermore, the locations to click may depend on all sorts of non-functional aspects, like the hardware (size of the screen), or user preference.

1.2 Bug, Error, Fault, Failures

These terms are often used, including by myself, interchangebly. However, in some of the discussions we would need to make the distinction between a fault, and its observable effect. So let me give the definition of those terms, at least for reference.

A *fault* in software is a mistake you make in the code. This mistake causes the software to move to a wrong state, which is termed *error*. This error may be observed by a user e.g. as the software gives a wrong answer, or by crashing, etc. This observed effect is called software *failure*. 'Bug' a popular term with no precise definition; we can use it if we do not really care for the distinction between the other three terms.

The important thing with those terms is that testing is aimed to find errors. But to actually fix those errors you still need to locate the faults that cause them. Locating faults is not always easy.

1.3 Test Level

People usually distinguish between unit testing, integration testing, and system testing. Generally, you can think a software to consist of components, which in itself are softwares, and may thus consist of lower level components. So you have components of various levels.

It is a good idea to test your low level components, e.g. your classes, in isolation. This is what people usually mean with *unit testing*. However, to test e.g. a class in isolation means that you would have to make some assumptions on how this class is used. In general it is hard to anticipate all possible ways this can happen, even in the context of your own application. That means, when the classes are composed to form a higher level component, new errors can be made. Therefore it is also useful to test your intermediate level components. This called *integration testing*. Finally, you would want to test the highest integration level, which is your the whole software itself. This is called *system testing*.

Errors are more easily found and fixed at the unit level. If they leak out to e.g. the system testing level, figuring out where the originating faults are is usually much harder (thus much more expensive). Therefore, it makes sense to invest in extensive unit testing.

1.4 Test Adequacy

Although testing is inherently incomplete, it is still the most commonly used approach to guard the quality and reliability of our software. So, from this pragmatic perspective it is valid to ask when testing can be considered as adequate.

Suppose we define one or more 'test coverage requirements'. An example of such a requirement is that our testing (thus, executing our test suites) have to pass every line of the SUT's code. Furthermore, we want these requirements to be *measurable* and *quantifiable*, so that we can infer how much, e.g. in percentage, of these requirements are fulfilled. So, we could then say that our testing delivers 90% coverage. With respect to the given 'test coverage requirements', testing is adequate if it delivers sufficiently high coverage. Ideally you would want 100% coverage.

There are various coverage criteria, e.g. line coverage, statement coverage, branch coverage, state coverage etc. Importantly, you should realize that no coverage criterion can provide a complete guarantee on the correctness of an SUT. If you use a weak criterion, you can finish testing quicker, but your chance of overlooking errors is also higher. On the other hand, picking a strong criterion means that you have to test more, which also increases your cost. From the pragmatic perspective, you would have to find a combination that balances the quality you aim for, and the resources that you have.

Several commonly used coverage criteria:

- *Function coverage*

  Suppose it is possible to view the SUT as a collection of 'functions'. E.g. if the SUT is a Java application, we can use its collection of methods.

  The *function coverage* of testing is the percentage of the functions from that collection that got executed by the testing.

  This is a very abstract concept of coverage, as it does not imply that every line in every function has been tested.

- *Line coverage*

  The percentage of the lines in the SUT source code which are visited during the testing.

- *Branch coverage*, also known as *descision coverage*.

  A branch statement is a statement like if-then-else and loops; it allows an execution to branch out to follow different paths. If we have branches in our program, ideally we should test all of them.

  The branch coverage of testing is the percentage of the branches in the SUT that are visited during the testing.

There are various tools available for measuring those coverage criteria. E.g. for Java you have Emma, Corbetura, and Clover.

Of the above criteria, branch coverage is the strongest of course. E.g. consider this example:

```
P(x,y) {
  if (even(x) && y>999) return x ;
  else                  return 0 ;
}
```

A single test case is sufficient to give 100% line coverage; but you would only get 50% branch coverage.

Branch coverage can be further strengthened by looking at the individual 'elementary conditions' that make up the guards of our branch statements. E.g. in the above example two test cases are sufficient to give 100% branch coverage, e.g.:

1 : `P(0,1000)`
2 : `P(0,0)`

However, there are actually two ways the if-then-else above is diverted to its else-branch: (1) because $\texttt{x} \leq \texttt{999}$, and (2) because `x` is odd. We have not yet tested the last scenario; from this perspective the above test cases are inadequate.

The condition `even(x)` && `x>999` can be seen as consiting of two elementary boolean conditions: `even(x)` and $\texttt{x} > \texttt{999}$. There is a stronger concept, called *condition/decision coverage* that requires that each elementary condition has to be tested, for both its true and false scenarios. Under this criterion, at least 4 test cases would be needed to give 100% coverage, e.g.:

1 : `P(0,1000)`
2 : `P(1,1000)`
3 : `P(0,0)`
4 : `P(1,0)`

Yet another variation of this is the *modified condition/decision coverage* (aka MC/DC). Though the idea is not difficult, I find it a bit difficult to explain. It requires that every elementary condition C should be tested for both its true and false scenarios, and furthermore by fixing the values of the other elementary conditions, and such that varying C will also vary the value of the whole condition:) Perhaps it is easier to explain this with an example. Consider again the program `P` above. Just these 3 test cases are needed to give 100% MC/DC coverage:

1 : `P(0,1000)`
2 : `P(1,1000)`
3 : `P(0,0)`

Here is a table showing the value of each elementary condition, and that of the whole condition per test case:

	test case	`even(x)`	`y>999`	the whole "`even(x)` && `y>999`"
1 :	`P(0,1000)`	T	T	T
2 :	`P(1,1000)`	F	T	F
3 :	`P(0,0)`	T	F	F

Notice that cases 1 and 2 variate the condition `even(x)` while fixing the value of the condition `y>999`. Notice how the value of the whole condition varies along because we varied the value of `even(x)`.

With test cases 1 and 3 with do the same, but this time we variate `y>999` while fixing `even(x)`.

Whereas the number of test cases needed for full condition/decision coverage is exponential with respect to the number of elementary conditions in a composite guard, the number of test cases for MC/DC is linear.

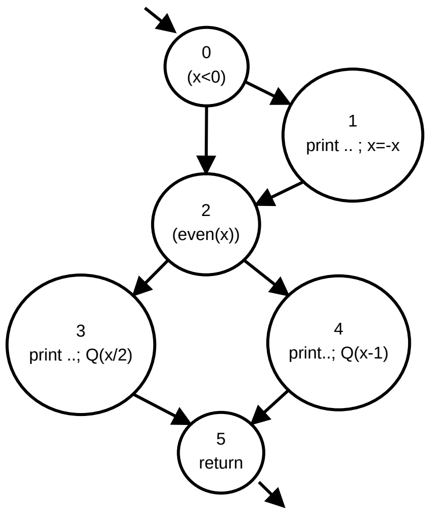
1.4.1 Path-based coverage

Consider now this program:

```
Q(x) {
  /* 0  */
  if (x<0)       { /* 1 */ print("negative!") ; x = -x ; }
  /* 2 */
  if (even(x))   { /* 3 */ print(".") ; Q(x/2) ; }
  else           { /* 4 */ print(".") ; Q(x-1) ; }
 /* 5 */
}
```

Two test cases are sufficient to give full branch coverage: `Q`$(-$`1`$)$ and `Q(0)`. But with just these we have not explored all possible control paths through the program, which, depending on your situation, may be considered as insufficient.

Let us abstractly view a program as a so-called *control flow graph* (CFG). The CFG of the above program is shown below. As the name says, it abstractly describes how the control flows within a program.



The nodes in the graph represents a sequential *block* `Q`. Such a block is a maximal segment of elementary statements in `Q` that execute sequentially. Furthermore:

- No statement in the middle of the block should jump/branch out from the block.
- No statement in the middle of the block should be the target of a jump from some block. Only the start of the block can be a jump target.

A block can be quite long, but it can also be vert short, consisting of just a single expression or elementary statement. The program `Q` has 6 blocks, I will number with 0..5. I have marked the start of each block in the program `Q` with a comment, as you can in the code above.

A CFG describes how the control within `Q` flows from one block to another. So if there is an arrow from block b to c, it means that after doing b, the program *may* continue to c. If it is the only outgoing arrow from b, then it *will* proceed to c. If we have e.g. two arrows from b, going to c and d, it means that after doing b the program may branch to either d or e. CFG's abstraction does not however tell when the program will choose one branch or the other.

A CFG would furthermore has a single entry node, that corresponds to the starting block of the program. It may have multiple exit node. Because of the maximality property we impose in the definition of 'block', no node can have exactly one incoming arrow and one outgoing arrow, since the block represented by this node wouldn't be maximal.
What the CFG above does tell us is that `Q` has four control paths, from the program's start to its end:

$$[0,1,3,5]$$
$$[0,1,4,5]$$
$$[0,2,3,5]$$
$$[0,2,4,5]$$

From this perspective, at least four test-cases would be needed to cover all those paths, e.g.:

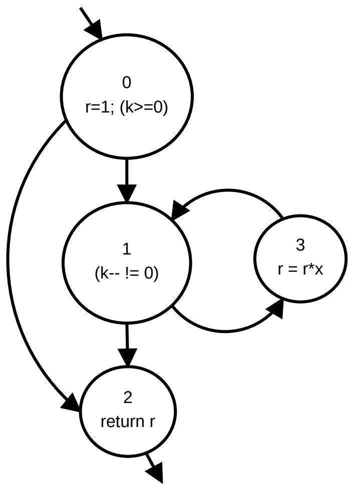
1 : `Q`(-2)
2 : `Q`(-1)
3 : `Q`(2)
4 : `Q`(1)

Roughly, path coverage is the percentage of the control paths through the CFG which are passed during the testing —we will refine the definition later. Full path coverage obviously implies full branch coverage, but not the other way around, as have been demonstrated above.

As a side note, we can also express statement and branch coverage in terms of CFG. *Node coverage* (or 'block coverage'), which is the percentage of the nodes in the CFG which are visited during the testing. This corresponds to statement coverage. *Edge coverage* is the percentage of the edges (arrows) in the CFG which are passed during the testing. This corresponds to branch coverage.

Consider now this program with a loop, with its CFG:

```
power(float x, int k) {
  int r = 1 ;
  if (k>=0) {
    while (k-- != 0) r = r*x ;
  }
  return r ;
}
```



Due to the cycle in the CFG, there are now infinite number of paths through the graph. So, we cannot aim towards covering all of them. What we can do instead is to require that every cycle should be iterated at least once, and that every non-cyclic combination in the paths should be covered. This is a bit vague, so let's make this more precise.

Let's begin by defining what a 'path' is. Let a CFG G be described by a pair (N, E) where N is its set of nodes, and $E : N \to Pow(N)$ describes the arrows, such that $E(x)$ gives us the set of nodes to which x is connected to in the graph.

An *exit node* is a node with no outgoing arrow. A *starting node* is a node with no incoming arrow. G is assumed to have just one starting node.

A (control) path in G is just a sequence of nodes such that consecutive nodes are connected by an arrow. A path should contain at least one node. If σ is a path, its length is the number of nodes it has, minus 1:

$$length(\sigma) \;=\; \#\sigma - 1 \tag{1.1}$$

So, a path of length 0 consists of one node.

A path is *maximal* if it starts at G's starting node, and ends at an exit node. A path is a *cycle* if its length is at least 1, and it ends in the same node as its starting node. An *elementary cycle* is a clycle that contains no subcycle.

Now notice that the paths $[1, 3, 1]$ and $[3, 1, 3]$ represent the same elementary cycle in the CFG above. We just need one. It does not matter which one; and let's call it the representative elementary cycle.

A *prime path* in a CFG is either a maximal path that contains no cycle, or a representative elementary cycle in the CFG. For example, the program `Q` we saw before has 4 prime path. The program `power` has 3 prime paths:

$[0, 2]$, $[0, 1, 2]$, and $[1, 3, 1]$

In particular the third prime path represents a test goal requiring you have to come up with a test case that exercises the loop by iterating at least once, whereas the second one, as we will see later, requires you to come up with a test case that makes the loop in `power` to immediately terminate (no iteration).

The concept of 'prime path' is originally proposed by Amman and Offutt [1]; we deviate slightly from the original.

Prime path coverage is the percentage of the number of prime paths of the program which are passed during the testing. When a test case is executed, the execution will tarverse some control path in the program's CFG. Let's call this 'execution path'. If we assume the program terminates, an execution path is thus always maximal.

Roughly, an execution path τ covers a target path σ if after some prefix it then does as σ. But because a prime path can be a cycle, technically we have to be a bit more careful with the definition:

Definition 1.4.1 : Direct Coverage
A path τ directly covers a non-cycle path σ if σ is a subpath of τ. That is, if:

$$\tau \;=\; prefix +\!\!+\; \sigma' +\!\!+\; sufix$$

for some paths $prefix$ and $sufix$.

If σ is a cycle, it is covered by τ if there is a permutation σ' of σ which is a subpath of τ.
□

E.g. the execution path $[0, 1, 3, 1, 2]$ of the program `power` we saw before (iterating the loop once) directly covers the prime path $[1, 3, 1]$, but not the other two prime paths. In particilar, it does not cover $[0, 1, 2]$. For the latter really need an execution that skips the loop, in this case the path $[0, 1, 2]$ itself.

With the above definition, three test cases would be needed to give a full prime path coverage on `power`. E.g.:

1 : `power(2,−1)`
2 : `power(2,0)`
3 : `power(2,1)`

Unfeasible paths and detour

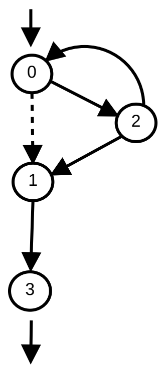
Some prime paths may actually be impossible to cover by real executions. E.g. if we require a pre-condition `k≥0` on the program `power` shown before, then it is impossible to cover the path $[0, 2]$. Such a path is called *unfeasible*.

If despite your effort you still can't get a full prime path coverage, and you suspect that it might be because some of the remaining paths are actually unfeasible, you may consider to fall back to a weaker criterion. You should use the information with care though, since by turning to a weaker criterion you risk overlooking some real errors.

Definition 1.4.2 : Detour
An execution path τ is said to cover a path σ by detour, if all nodes in σ appear in τ, and they appear in the same order.

Another way to say this that σ can be obtained from τ by deleting some nodes in τ.
□

For example in the CFG below, that the path $[0, 2, 1, 3]$ would cover $[0, 1, 3]$ by detour:



Path coverage by detour is not the same as node nor branch coverage; this can be shown by the above example:

- A single execution $[0, 2, 1, 3]$ is enough to give a full node coverage.
- Two executions paths $[0, 1, 3]$ and $[0, 2, 0, 2, 1, 3]$ gives a full branch coverage. The same execution will also give full direct prime path coverage (there are three prime paths to cover).
- If detour is allowed, just the executions $[0, 2, 1, 3]$ and $[0, 2, 0, 2, 1, 3]$ would give us full prime path coverage.

Now suppose that we suspect the branch from 0 to 1 is actually impossible (indicated by the dashed line in the picture above). Full direct prime path coverage would then be unfeasible. Full branch coverage is also unfeasible.

Node coverage is a possibility, but it does not force you to test the cycles in your program. E.g. the execution path $[0, 2, 1, 3]$ gives you a full node coverage, without ever executing the cycle $[0, 2, 0]$.

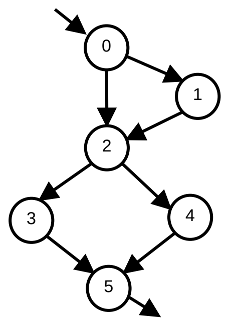
Full prime path coverage by detour is still feasible, and would be better than just taking node coverage, because it will not allow you to skip cycles (just $[0, 2, 1, 3]$ is not sufficient for full prime path coverage by detour).

Notice that 'detour' has been defined in such a way that you can avoid passing the arrow connecting two nodes i and j by going trough an alternate route from i to j.

1.4.2 Linearly Independent Paths

The number of prime paths is exponential. So, if you have a program with e.g. 10 in-then-else in a series, you'll have a lot of work to do. In practice such a program does not occur very often, so prime path is not that bad.

An often suggested alternative is to use linearly independent paths. A set Σ of paths are linearly independent to each other if every path $\sigma \in \Sigma$ has at least one unique edge that is not present in all the other paths in Σ. For example, consider this CFG:



The paths $[0, 2, 3, 5]$ and $[0, 1, 2, 4, 5]$ are linearly independent to each other; so two test cases are sufficient to cover them. But so are the set of these paths:

$$[0, 2, 3, 5],\ [0, 1, 2, 3, 5],\ [0, 2, 4, 5]$$

for which three test cases would be needed. You already know the popular McCabe complexity number. Now this number gives an upper bound to the size of the set of linearly independent paths in an CFG.

McCabe complexity number is $e - n + 2$, where e is the number of edges in the CFG, and n is the number of its nodes. It is much less than the number of prime paths. Though from my perspective it seems to be just slightly stronger than branch coverage.

1.5 White box and black box testing

There is nothing special about writing test cases. They are just programs. E.g. a test case for the program `power` could look like this:

```
power_test() {
   assert power(2,0)==1 ;
   assert power(2,1)==2 ;
   assert power(2,3)==8 ;
   assert power(2,-1)==1 ;
}
```

But how do we come up with these tests in the first place? Remember that you would want your tests to deliver full coverage. The obvious source of information is the source code of your SUT. If that is how you infer your test cases then we call your testing *white box testing*.

However, source code is not always available. And even if it is, you may deliberately decide not to look at it, e.g. because it would flood you with too many details. Your testing is then called *black box testing*.

1.5.1 Classification tree for black box testing

A practical approach you can use for black box testing is by using classification trees [4]. Consider a hypothetical program to register grades for students. The program has this header:

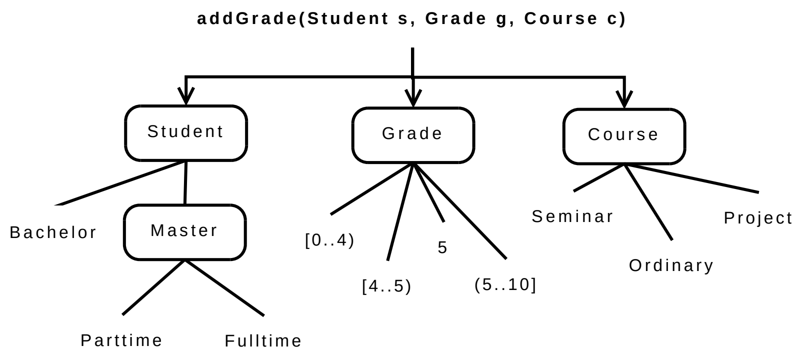
```
addGrade(Student s, Grade g, Course c)
```

We say that this program has three input domains: the domain of students, the domain of grades, and the domain of courses. By e.g. the 'domain' of students we simply mean the set of all possible students we can give to this program.

Rather than testing this program with arbitrarty combinations of students, grades, etc, we try to identify logical partitions of each domain into subdomains. For example, it may make sense, for testing purpose, to distinguish between bacelor and master students.

Remember that you cannot look into the source code to come up with a sensical partitioning. So, you have to rely on other sources, e.g. specifications or software documentation.

The partitioning can be carried out recursively as needed. For example, we could partition the domains of `addGrade` as shown in the tree below:

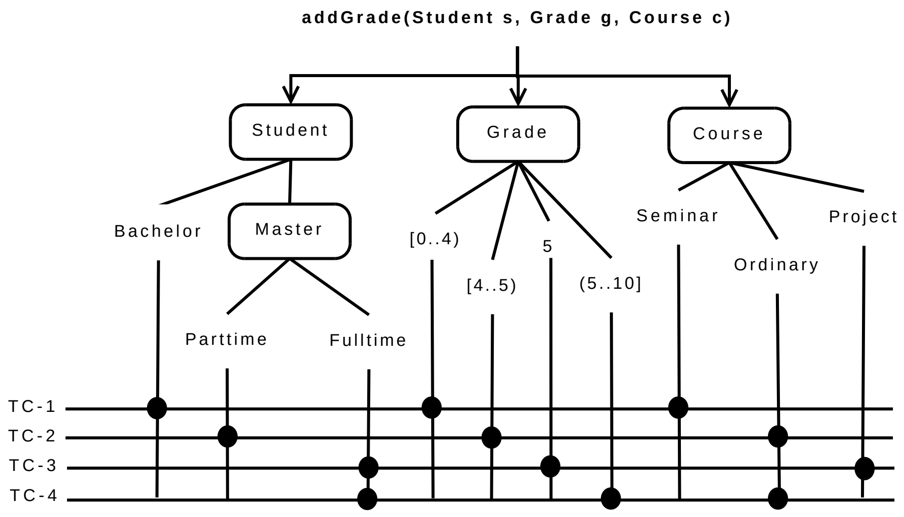


The tree above is called *classification tree* (CT). The boxes and the leaves represent domains or subdomains. In CT jargon they are called 'classes'.

For each leaf-class you would need to supply at least one value to be used as an input for `addGrade`. So, you would need to supply at least one bachelor student, one parttime master student, one fulltime master student, etc.

Obviously, by combining supplied inputs from different leaf-classes you can produce various test cases. You can generate all combinations to give full combinatorial CT coverage (`addGrade` has in total 36 combinations). But since the total number of combinations explodes fast if you have a large classification tree, you may opt to just generate combinations such that every class would be tested at least once to give full CT class coverage.

There is also a tool called Classification Tree Editor (CTE) from `systematic-testing.com` that allows you to generate those combinations, or to edit them manually if need to. Desinging test cases using CTE looks more or less like the picture below:



Each horizontal like describe a single test case. The black circles on the line specify the input combinations the test case take. E.g. the first test case take a bachelor student, a grade in the range [0..4), and a seminar as inputs for `addGrade`.

The picture above would then describe a set of four test cases. By the way, this suite would give you full TC class coverage (but not full combinatorial coverage, of course).

1.6 Regression

A software's life does not ends at its delivery. Most software is so complicated that it is not feasible to deliver it bug free right on. So, after delivery maintenance continues to fix bugs found afterwards. The software's owner and customers may request changes in features or functionalities, and adding new ones. This again triggers modifications in the software.

Now, before a new version of the software is released into the production we would need to test it again, to make sure that it would be as reliable as the old one. This is called *regression testing.* We do not generally write the needed test cases from the scratch —that would be very inefficient. Instead, we would want to reuse existing test suite, as far as it can be reused, and complementing it with new test cases as needed.

Obviously if you add a new feature then you likely need to provide additional test cases. The issue is then just the same as writing test cases in general.

Selecting the reusable part of the old test suite is however another issue.

1.6.1 Test cases selection

Over its life time a software can accumulate a huge amount of test cases. So simply re-executing the whole test suite can take a very long time, e.g. hours, or even days. Errors leaking to the regression phase (and found) are expensive. Not only you have to fix them, but you would have to go through another regression round to make sure that this time nothing else is broken.

Regression can be made faster if we apply test case selection. The obvious thing to do is to just select those test cases that exercise the modified components of the software —e.g. you can take 'classes' as 'components'. However some components may be exercised very often. E.g. if to you a software you have to login first, then all test cases at the system testing level will exercise the login component. The above approach would simply select the entire test suite for regression, even if you only change one branch in this component.

Let's first define some concepts. Earlier we have said that we model an SUT as a program with possibly multiple operations. For simplicity let us now take an even simpler view. Our SUT is just a single procedure. It takes a single input parameter, returns a value as its response.

A test case t for such an SUT is modelled by a pair (i, o), where i is the input it gives to the SUT, and o is the expected output. The SUT fails the test if its actual output is unequal to this o.

Let P be the SUT, and P' be its modified version. The response produced by P on input i will be denoted by $P(i)$. Let $S = (S_{pre}, S_{post})$ be the hypothetical specification of P. Similarly, let S' be the hypothetical specification of P'. They are hypothetical because in reality people may not have specifications; or if they do, the specifications may be only partial.

S_{pre} specifies P's pre-condition. It is a predicate specifying valid inputs of P. The S_{post} specifies P's post-condition. It is a predicate specifying correct outputs. A test-case $t = (i, o)$ is *consistent* with a specification S if $S_{pre}(i)$ and $S_{post}(o)$ are both true. All test cases for a program P should of course be consistent with P's specification.

Let T be P's test suite. We assume that P has been tested against T and have passed all the test cases there. For regressing test, we want to indentify those test cases from T which can be reused to test P'.

Now, those test cases from T which are no longer consistent with S' (the specification of P') are *obsolete.* We don't want them, but unfortunately there is no effective procedure to identify them (halting problem). That's not what we are going to do anyway; but we do need the concept for the theory.

A test case $t \in T$ if *fault revealing* (with respect to P') if $t = (i, o)$ is non-obsolete and it causes P' to fail [6]. That is, $P'(i) \neq o$. However, notice this:

$$P'(i) \neq o$$

$$= \{\ P \text{ has passed } T\text{, including } t\text{; so } P(i) = o\ \}$$

$$P'(i) \neq P(i)$$

So, this gives the following theorem:

Theorem 1.6.1 : FAULT REVEALING BY OUTPUT-DIFFERENCE
An non-obsolete test case $t = (i, o)$ of P is fault revealing with respect to P' if and only if P and P' produce different outputs on the input i.
□

The ideal goal of *test selection* is to find a subset $Ideal \subseteq T$ consisting of the test cases that are fault revealing with respect to P'. Now we can just execute T' and save the computation time Δ that would otherwise be spent on executing $T/Ideal$. However, determining or approximating $Ideal$ also costs you computation time Σ. So the net gain is $\Sigma - \Delta$. All the effort for doing test selection is only economical if this net gain is > 0.

Another thing to point out is that finding the exact subset of fault revealing test cases of T is in general undecidable (due to the halting problem)[1]. However, we can estimate it.

So, for a practical test selection algorithm, you can talk about its 'precision' and 'safety'. Suppose this algorithm come up with a subset $T' \subseteq T$. Its *precision* is:

$$1 \; - \; \frac{\#(T'/Ideal)}{\#(T/Ideal)} \quad \text{, expressed in percentage, and 100\% if } T/T' = \emptyset. \tag{1.2}$$

This simply express how much 'noisy' test-cases that are selected along. Precision 100% implies no noisy test-case is selected.

A selection algorithm is *safe* if it manages to select all test cases from $Ideal$. You would want an algorithm that is safe, whereas precision has to be balanced against the net gain.

Now, a consequence of Theorem 1.6.1 is a non-obsolete test case $t = (i, o)$ is fault revealing if and only if t passes a part of P which is modified or deleted in P'; because that is the only way P' can end up with a different output on the same input i. Such a test case is called *modification traversing*. The above observation gives these theorems:

Theorem 1.6.2 : FAULT REVEALING BY MODIFICATION TRAVERSING
An non-obsolete test case of P is fault revealing with respect to P' if and only if it is modification traversing.
□

Theorem 1.6.3 : MODIFICATION TRAVERSING ALGORITHM
This selection algorithm safe.

$$\text{take as } T' \;=\; \{t \in T \mid t \text{ is modification traversing}\}$$

□

These justify the intuition that you may already have all along.

Let G be the CFG of P. In principle you can instrument P, so that when you run a test case t on P you know which nodes in G are passed by the execution. Therefore, we can implement this function:

$$testCasesPassing(n) \;=\; \{t = (i, o) \in T \mid P(i) \text{ passes the node } n\}$$

It returns the subset of T that pass n.

So, if we can find all nodes in G which are modified (or deleted) in P', then we are done. Because then T' in Theorem 1.6.3 is just:

$$T' \;=\; (\cup n : n \in G \text{ and } n \text{ is modified} : testCasesPassing(n))$$

I will show you now an algorithm by Rothermel and Harrold [7]. It is a test selection algorithm. However, i find it easier to understand if you see it as an algorithm to find modified nodes. Once

[1]Take $P =$ `skip`. P' is just P appended with an arbitrary program Q. P obviously terminates on input 0. Take as spec S': P' terminates. So, 0 is fault revealing wrt P' iff Q terminates on 0. This maps halting on-0 problem to test selection.

you find them, the above formula gives you the test cases. So, I will give you a slightly adapted version of the original one.

We will extend our CFG a bit. A node that has multiple outgoing arrows represents a descision node. If we first desugar conditional statements with multiple alternatives like `case` to cascades of `if-then-else`, all decision nodes will then only have two out-arrows. We will them with `T` and `F`, to mark the 'then' branch and the 'else' branch. Arrows from non-decision nodes have empty label ϵ. So, for example:

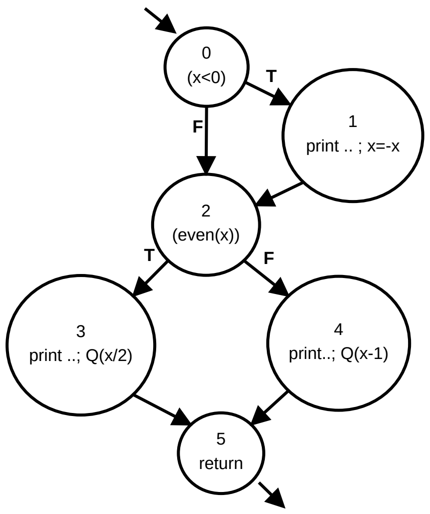


Figure 1.1 shows Rothermel and Harrold's algorithm. To use it, we first setup the inputs G, G', and T. Then call the method `main`.

The main work is done by the method `compare`. In principle, this identifies all modified nodes, and put them in the set `modified`. For each modified node n, `main` then add the subset of T that pass it to T', and finally return T'.

The function `identical(n,n')` checks whether the statements associated to the node n respectively n' are textually identical (comments and white-space are removed). Else you can also check if their byte code instructions are identical. If they are not identical, then n has been modified (or deleted) in P'.

Notice that the above algorithm may not find all modified nodes; but this is not neccessary either. If a node o is only reachable in P through the node n, $testCasesPassing(o)$ must be a subset of $testCasesPassing(n)$. If we have already found that n is modified, $testCasesPassing(n)$ will be later added to T'. So, there is no point to consider o. The algorithm avoids this indeed.

1.7 References

- P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, 2008

- G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997

```
class SelectTests {

   // Inputs for the algorithm:
   CFG G = ...  ;  // the CFG of P
   CFG G'= ...  ;  // the CFG of P'
   Set<testcase> T = ... ; // T is a test suite of P; P has passed testing against T


   Set<testcase> main() {
      compare(G.root, G'.root) ;
      T' = ∅ ;
      for (n : modified) T' = T' uni testCasesPassing(n) ;
      return T'
   }


   // Working data structure:
   Set<Node> modified = ∅  ;

   compare(Node n, Node n') {
      mark n as n'-visited ;
      if ¬ identical(n,n') {
         modified = modified.insert(n) ;
         return ;
      }

      // else continue:

      for (o : G.next(n)) {
         lab = the label of the arrow n  →  o ;
         o' = the successor of n' in G', such that n' → o' has the label 'lab' ;
         if (o is  not marked as o'-visited)
            // recurse:
            compare(o,o') ;
      }
   }
}
```

Figure 1.1: Rothermel and Harrold selection algorithm (slightly adapted).

Chapter 2

Hoare Logic

Hoare logic is a simple logic to prove the correctness of sequential imperative programs. It often forms the basis for other logics for imperative programs.

Programs are specified by *Hoare triples* like:

$$\{* \ \#\mathtt{S} > 0 \ *\} \quad \mathtt{getMax(S)} \quad \{* \ \mathtt{return}{\in}\mathtt{S} \wedge (\exists \mathtt{x} : \mathtt{x}{\in}\mathtt{S} : \mathtt{return}{\geq}\mathtt{x}) \ *\}$$

Generally a specification takes the form $\{* \ P \ *\}S\{* \ Q \ *\}$ where P and Q are predicates, and S is the program being specified. P is also pre-condition, and Q post-condition.

It means that if S is executed in a state satisfying P, on termination its state will satisfies Q.

Note that in the above definition termination is assumed. You can also require that such a specification should also imply that S terminates (when executed from P). Requiring termination is obviously stronger.

Hoare triple interpretation where termination is assumed is also called *partial correctness* interpretation, and otherwise it is called *total correctness* interpretation.

2.1 Examples of Hoare logic's inference rules

This rule allows you to infer a Hoare triple with a weaker post-condition:

Rule 2.1.1 : Post-condition Weakening

$$\frac{\begin{array}{c} \vdash \ Q \Rightarrow Q' \\ \{* \ P \ *\} \ S \ \{* \ Q \ *\} \end{array}}{\{* \ P \ *\} \ S \ \{* \ Q' \ *\}}$$

Analogously, you can strengthen a pre-condition. Hoare triples are also conjunctive and disjunctive.

With this rule you can split the proof of a sequential composition of two statements:

Rule 2.1.2 : SEQ

$$\frac{\begin{array}{c} \{* \ P \ *\} \ S_1 \ \{* \ P' \ *\} \\ \{* \ P' \ *\} \ S_2 \ \{* \ Q \ *\} \end{array}}{\{* \ P \ *\} \quad (S_1; \ S_2) \quad \{* \ Q \ *\}}$$

Dealing with sequential composition is very important, because that is probably the most important way to compose your programs from elementary statements.

Unfortunately the above rule requires you to come up with some intermediate P' and the rule does not give much clue what it should be. There is no general algorithm to infer P', though in some situation we can do it.

If given a program S and a post-condition Q there is an algorithm to calculate a sufficient pre-condition for S to establish Q, then the above problem is solved.

2.2 Weakest pre-condition calculation

Let $\mathsf{wp}\ S\ Q$ denotes the weakest pre-condition for the program S to realize the post-condition Q. It can be characterized by this property:

$$\{*\ P\ *\}\ S\ \{*\ Q\ *\} \quad = \quad P \Rightarrow (\mathsf{wp}\ S\ Q) \tag{2.1}$$

In literature people also distinguish between wp and wlp. The latter stands for *weakest liberal pre-condition*, which defined the same as wp, except that we assume termination.

Here is how we can calculate the weakest pre-condition of some simple statements:

1. $\mathsf{wp}\ x := e\ Q \quad = \quad Q[e/x]$
2. $\mathsf{wp}\ (S; T)\ Q \quad = \quad \mathsf{wp}\ S\ (\mathsf{wp}\ T\ Q)$
3. $\mathsf{wp}\ (\ \texttt{if}\ g\ \texttt{then}\ S\ \texttt{else}\ T)\ Q \quad = \quad (g \Rightarrow (\mathsf{wp}\ S\ Q))\ \wedge\ (\neg g \Rightarrow (\mathsf{wp}\ T\ Q))$

Analogous 'rules' apply for wlp.

Unfortunately, there is no wp nor wlp rules for loops and recursions.

By combining wp rules we can in principle calculate the weakest pre-condition of any composite statement that does not contain loops nor recursions.

2.3 Inference rule for loop

Here is the inference rule to handle loop. It assumes the partial correctness interpretation; so with it you still have not proven that the loop terminates. To prove termination you would need to prove some additional conditions.

Rule 2.3.1 : Loop

$$\frac{\begin{array}{c} \vdash\ P \Rightarrow I \\ \{*\ I \wedge g\ *\}\ S\ \{*\ I\ *\} \\ \vdash I \wedge \neg g \Rightarrow Q \end{array}}{\{*\ P\ *\}\ \ \texttt{while}\ g\ \texttt{do}\ S\ \ \{*\ Q\ *\}}$$

Unfortunately the rule requires you to come up with an I. This predicate I is also called *loop invariant*. There is no general algorithm to infer I.

Example

Prove the validity of this specification:

$$\{*\ \mathtt{i}{\geq}0\ *\}\ \ \texttt{while i>0 do i} := \mathtt{i}{-}1\ \ \{*\ \mathtt{i} = 0\ *\}$$

Answer: using the pre-condition $\mathtt{i}{\geq}0$ itself as the invariant I will do. The conditions are then trivial.

Example

The following program sums the elements of the array a. Prove its correctness:

$$\{*\ \texttt{true}\ *\}\ \ \mathtt{i} = \#\mathtt{a};\ \ \texttt{while i>0 do}\ \{\mathtt{i} := \mathtt{i}{-}1;\ \mathtt{s} := \mathtt{s}{+}\mathtt{a}[\mathtt{i}]\}\ \ \{*\ \mathtt{s} = (\Sigma \mathtt{j} : 0{\leq}\mathtt{j}{<}\#\mathtt{a} : \mathtt{a}[\mathtt{j}])\ *\}$$

Answer: using this as invariant:

$$0{\leq}\mathtt{i}{\leq}\#\mathtt{a}\ \ \wedge\ \ \mathtt{s} = (\Sigma \mathtt{j} : 0{\leq}\mathtt{j}{<}\mathtt{i} : \mathtt{a}[\mathtt{j}])$$

Prove the conditions yourself.

Example

The following program checks if a boolean array `b` contains a `true`. Prove its correctness:

$\{*\ \texttt{true}\ *\}$

$\texttt{i} = 0;\ \texttt{found} := \texttt{false}$

$\texttt{while i<\#a} \wedge \neg\texttt{found do}\ \{\texttt{found} := \texttt{b[i]};\ \texttt{i} := \texttt{i+1}\}$

$\{*\ \texttt{found} = (\exists \texttt{j} : 0 \leq \texttt{j} < \texttt{\#b} : \texttt{b[j]})\ *\}$

Answer: using this as invariant:

$0 \leq \texttt{i} \leq \texttt{\#b} \quad \wedge \quad \texttt{found} = (\exists \texttt{j} : 0 \leq \texttt{j} < \texttt{i} : \texttt{b[j]})$

Prove the conditions yourself.

2.4 Invariant as abstraction

Loop invariant can be seen as an abstraction of a loop. Knowing the invariant gives you sufficient information to know the goal of the loop, which you can infer from $I \wedge \neg g$, without having to look at the loop itself, including its various implementation details.

E.g. in the third example above, the loop is optimized to break as soon as the right element has been found. If you just want to know what the loop mainly does, such a detail is less important. The invariant gives you in the case clearly a more abstract view.

2.5 Rule for proving loop termination

To prove that a loop terminates the idea is to come up with an integer expression m (which can be interpreted over your program states) called *termination metric*. This m should be such that its value decreases at each iteration. However, your invariant implies that m has a lower bound, e.g. 0. These imply that you cannot iterate forever, as m would then breach its lower bound, which is a contradiction.

This is formally captured by this strengthened version of loop rule:

Rule 2.5.1 : Loop with Total Correctness

$$\frac{\begin{array}{c} \vdash P \Rightarrow I \\ \{*\ I \wedge g\ *\}\ \ (C := m; S)\ \ \{*\ m < C\ *\} \\ \vdash I \wedge g \Rightarrow m > 0 \\ \vdash I \wedge \neg g \Rightarrow Q \\ \{*\ I \wedge g\ *\}\ \ S\ \ \{*\ I\ *\} \end{array}}{\{*\ P\ *\}\ \ \texttt{while}\ \ g\ \texttt{do}\ S\ \ \{*\ Q\ *\}}$$

In general m can also be expression that retruns a value from some domain D, equiped with a well-founded relation $\prec$. The condition that m has a lower bound can be replaced with a requirement that I implies that m will be closed within D.

There unfortunately no general algorithm to infer m; so you have to come up with one yourself.

Example

Prove that this loop terminates:

$\{*\ \texttt{i} \geq 0\ *\}\ \ \texttt{while i>0 do i} := \texttt{i}-1\ \ \{*\ \texttt{i} = 0\ *\}$

Answer: we have used $\texttt{i} \geq 0$ the invariant I to prove the validity of the specification in partial correctness. We will use that invariant.

As the termination metric, the expression `i` will do.

Example

Suppose we have a bag of red and blue candies. Each day you can: (1) take one red candy from the bag, or (2) take a more delicious blue one, but then you have to put one new red candy in the bag. Prove that eventually the bag will be empty, regardless your daily choice of actions.

We will prove this by writing a program that simulates the process:

```
{* r≥0 ∧ b≥0 *}

 while r+b > 0 do {
   if r>0 → r := r − 1
   [] b>0 → {b := b − 1; r := r + 1}
   fi
  }

{* r=0 ∧ b=0 *}
```

The `if` above is non-deterministic. So, if the specification is valid we will end up with both no more candy in the bag (`r=0 ∧b=0`), despite the non-deterministic choices made by the `IF` at each iteration.

To prove the above specification, these invariant and termination metric will do:

I : `r≥0 ∧ b≥0`

m : `r + 2b`

2.6 Abstract model of programs

An *abstract model* of an artifact abstractly models the artifact. The adjective 'abstract' is actually a bit superflous because a 'model' is, by the word's meaning, already abstract. Abstract models are useful for explaining and understanding programming logic; in our case Hoare logic.

Let us abstractly model the *state* of program by a record that maps the program's variables to their values in that state. E.g. $\{x{=}0, y{=}0\}$ represents a state of some program with two variables x, y, and in that state the value of x is 0, and y is 9.

Let us for simplicity assume that we have a fixed set of program variables; we name this set Var. All statements we will discuss are assumed to operate on states over Var. Let Σ be the set of all possible states over this Var.

An expression can be abstractly modelled by a function e of type $\Sigma \rightarrow val$, where val is the type of the value returned by the expression. A *predicate* is just an expression that returns a boolean value.

For example the predicate $x{>}y$ is modelled by the function $(\lambda s.\ s.x > s.y)$.

To keep the notation light, I will usually use the term 'predicate' and its abstract model interchangebly. When I name a predicate P, I often use the same symbol to denote its model. You'll have read it from the context which one is meant. The same goes with other program elements, e.g. state or expression.

A predicate P induces a set, namely the set of all states that satisfies it. If we denote this set as χ_P, it is:

$$\chi_P \quad = \quad \{s \mid P\ s = \mathtt{true}\}$$

Conversely, if we know the set, it defines the corresponding predicate. Therefore I will also the term predicate and the set of states that it induces interchangebly.

In terms of sets, conjunction and disjunction of two predicates correspond to set intersection and union. For implication:

$$P \Rightarrow Q \quad = \quad (\mathtt{true}/P) \cup Q$$

The validity of an implication corresponds to subset. Let us write $\models P$ to mean that P is a valid predicate. That is, it is true on all states in Σ.

$$\models (P \Rightarrow Q) \quad = \quad P \subseteq Q$$

A program can be abstractly modelled by a function of type $\Sigma \to \Sigma$. However, with this we cannot model a non-deterministic program. To allow the latter, we take this model:

$$\Sigma \to Pow(\Sigma)$$

where $Pow(\Sigma)$ is the power set of Σ. Let S be a program; then $S\ s$ is the set of all possible end-states if it is executed on s.

In this discussion we will just assume that our programs always terminate; so $S\ s$ is never empty.

For example the simple statement `x++` is modelled by:

$$(\lambda s.\ \ \{\{x{=}s.x{+}1,\ y{=}s.y\}\})$$

Whereas the following is a model of a statement that non-deterministically chooses between doing a skip or $\mathtt{x}++$:

$$(\lambda s.\ \ \{s,\ \{x{=}s.x{+}1,\ y{=}s.y\}\})$$

Models of 'if' and sequential compositions:

$$\texttt{if}\ g\ \texttt{then}\ S\ \texttt{else}\ T \quad = \quad (\lambda s.\ \texttt{if}\ g\ s\ \texttt{then}\ S\ s\ \texttt{else}\ T\ s) \tag{2.2}$$

$$S_1\ ;\ S_2 \quad = \quad (\lambda s. \bigcup \{S_2\ t \mid t \in S_1\ s\}) \tag{2.3}$$

Now we have enough to give an abstract model of Hoare triple:

$$\{*\ P\ *\}\ S\ \{*\ Q\ *\} \quad = \quad (\forall s :: s{\in}P \Rightarrow S\ s \subseteq Q) \tag{2.4}$$

And the weakest pre-condition is:

$$\mathsf{wp}\ S\ Q \quad = \quad \{s \mid S\ s \subseteq Q\} \tag{2.5}$$

Now that we have these models, we can justify the programming rules that we had. For example to justify the post-condition weakening rule:

$$\frac{\begin{array}{c}\vdash\ Q \Rightarrow Q' \\ \{*\ P\ *\}\ S\ \{*\ Q\ *\}\end{array}}{\{*\ P\ *\}\ S\ \{*\ Q'\ *\}}$$

In terms of models we have to show that for any $s{\in}P$, $S\ s \subseteq Q'$. The given specification implies that $S\ s \subseteq Q$. But the first condition above says that $Q \subseteq Q'$. So, we are done.

2.7 uPL

uPL is a simple imperative programming language we can use to study Hoare logic. Here are a summary of its features:

- It has basic control structures like `if` and `while`.
- It has primitive types like `bool` and `int`, arrays, and records.

  To simplify the logic, we will assume that uPL arrays are infinitely large. The indices go from $-\infty$ to $+\infty$.

- No global variables. A program can only access its own local variables and its parameters.
- Parameters are passed either by value or by a copy-restore protocol.

Here is an example of a uPL program that calculates the longest common prefix of two strings:

```
getLongestPrefix (N:int, s,t:char[]) : int
   { var break : bool ;
     var i : int ;
     break := false ;
     i := 0
     while ¬break ∧ i<N do {
        break := s[i]≠t[i] ;
        i := i+1
        }
     return i ;
   }
```

The syntax of uPL is given below.

The syntax of programs

⟨*program*⟩ ::= ⟨*header*⟩ ⟨*body*⟩

⟨*header*⟩ ::= ⟨*program-name*⟩ '(' ⟨*formal-parameter-list*⟩ ')' ':' ⟨*type*⟩

⟨*formal-parameter-list*⟩ ::= ⟨*empty*⟩
  | ⟨*formal-parameter*⟩ (',' ⟨*formal-parameter*⟩)*

⟨*formal-parameter*⟩ ::= `OUT`? `READ`? ⟨*var-decl*⟩

⟨*var-decl*⟩ ::= ⟨*var-name*⟩ (',' ⟨*var-name*⟩)* ':' ⟨*type*⟩

⟨*body*⟩ ::= '{' ⟨*locvar-decl-list*⟩ ⟨*statement-sequence*⟩ ⟨*return*⟩ '}'

⟨*locvar-decl*⟩ ::= `var` ⟨*var-decl*⟩

⟨*locvar-decl-list*⟩ ::= (⟨*locvar-decl*⟩ ';')*

⟨*return*⟩ ::= ⟨*empty*⟩ | ';' `return` ⟨*expr*⟩

⟨*statement*⟩ ::= `skip`
  | ⟨*assignment*⟩
  | '{' ⟨*locvar-decl-list*⟩ ⟨*statement-sequence*⟩ '}'
  | ⟨*if-then-else*⟩
  | ⟨*while-loop*⟩
  | ⟨*program-call*⟩

⟨*assignment*⟩ ::= ⟨*target*⟩ `:=` ⟨*expr*⟩

⟨*target*⟩ ::= ⟨*variable*⟩ | ⟨*array-element*⟩ | ⟨*record-element*⟩

⟨*statement-sequence*⟩ ::= ⟨*statement*⟩ (';' ⟨*statement*⟩)*

⟨*if-then-else*⟩ ::= `if` ⟨*expr*⟩ `then` ⟨*statement*⟩ `else` ⟨*statement*⟩

⟨*while-loop*⟩ ::= `while` ⟨*expr*⟩ `do` ⟨*statement*⟩

⟨*program-call*⟩ ::= ⟨*call*⟩ | ⟨*target*⟩ `:=` ⟨*call*⟩

⟨*call*⟩ ::= ⟨*program-name*⟩ '(' ⟨*actual-parameter-list*⟩ ')'

⟨*actual-parameter-list*⟩ ::= ⟨*empty*⟩ | ⟨*expr*⟩ (',' ⟨*expr*⟩)*

Available types

$\langle type \rangle$ ::= $\langle simple\text{-}type \rangle$ | $\langle record\text{-}type \rangle$ | $\langle array\text{-}type \rangle$ | $\langle type\text{-}name \rangle$

$\langle simple\text{-}type \rangle$ ::= `()` | `bool` | `int` | `char` | `string`

$\langle record\text{-}type \rangle$ ::= `record` '{' $\langle field\text{-}def \rangle$ (';' $\langle field\text{-}def \rangle$)* '}'

$\langle field\text{-}def \rangle$ ::= $\langle field\text{-}name \rangle$ (',' $\langle field\text{-}name \rangle$)* ':' $\langle simple\text{-}type \rangle$

$\langle array\text{-}type \rangle$ ::= ($\langle simple\text{-}type \rangle$ | $\langle record\text{-}type \rangle$) `[]`+

$\langle expr \rangle$ ::= $\langle constant \rangle$
  | $\langle variable\text{-}name \rangle$
  | $\langle array\text{-}element \rangle$
  | $\langle record\text{-}element \rangle$
  | $\neg\langle expr \rangle$
  | $\langle expr \rangle$ $\langle binary\text{-}operator \rangle$ $\langle expr \rangle$
  | '(' $\langle expr \rangle$ ')'

$\langle array\text{-}element \rangle$ ::= $\langle variable\text{-}name \rangle$ ('[' $\langle expr \rangle$ ']')+

$\langle record\text{-}element \rangle$ ::= $\langle expr \rangle$ '.' $\langle field\text{-}name \rangle$

uPL Operators

See the table below, listed in the decreasing order of their priority (so, the top row lists the highest priority operators). The L flag means that the corresponding operator is left-associative; R means that it is right associative; and LR means that the operator is both left and right associative. Operators with no L/R flag is not associative.

`^` (L)
`*` (LR)
`+` (LR), `-` (L)
`mod`, `div`, `max` (LR), `min` (LR)
$<, >, \neq, \leq, \geq$
$\wedge$ (LR)
$\vee$ (LR)
`=`

2.8 Some Commonly Used Inference Rules and Theorems of Predicate Logic

Rule 2.8.1 : Excluded Middle

$$\frac{-}{P \vee \neg P}$$

Rule 2.8.2 : Modus Ponens ($\Rightarrow$ Elimination)

$$\frac{P \qquad P \Rightarrow Q}{Q}$$

Rule 2.8.3 : Contradiction

$$\frac{P \qquad \neg P}{\texttt{false}} \qquad \frac{P \Rightarrow \texttt{false}}{\neg P}$$

Rule 2.8.4 : True Consequence

$$\frac{P = \texttt{true}}{P} \qquad \frac{\texttt{true} \Rightarrow P}{P}$$

Rule 2.8.5 : $\wedge$ Elimination

$$\frac{P \wedge Q}{Q} \qquad \frac{P \wedge Q}{P}$$

Rule 2.8.6 : Conjunction ($\wedge$ Introduction)

$$\frac{P \qquad Q}{P \wedge Q}$$

Rule 2.8.7 : ∨ Elimination

$$\frac{\begin{array}{l}\neg P\\ P \vee Q\end{array}}{Q}$$

Rule 2.8.8 : ∨ Introduction

1. $$\frac{\neg P \Rightarrow Q}{P \vee Q}$$
2. An easier version. But it is weaker, since you may fail to prove Q without the help of $\neg P$:

$$\frac{Q}{P \vee Q}$$

Rule 2.8.9 : Case Split

$$\frac{\begin{array}{l}P \Rightarrow Q\\ \neg P \Rightarrow Q\end{array}}{Q} \qquad \frac{\begin{array}{l}P_1 \vee P_2\\ P_1 \Rightarrow Q\\ P_2 \Rightarrow Q\end{array}}{Q}$$

Rule 2.8.10 : Specialization (∀-Elimination)

$$\frac{\begin{array}{l}P\ e\\ (\forall i : P\ i : Q\ i)\end{array}}{Q\ e} \qquad \frac{(\forall i : P\ i : Q\ i)}{P\ e \ \Rightarrow\ Q\ e}$$

Rule 2.8.11 : ∃ Introduction

$$\frac{P\ e}{(\exists i :: P\ i)} \qquad \frac{\begin{array}{l}P\ e\\ Q\ e\end{array}}{(\exists i : P\ i : Q\ i)}$$

Theorem 2.8.12 : Basic equalities of Boolean connectors

1. $\vdash \neg\neg P = P$
2. $\vdash P \vee Q = Q \vee P$
3. $\vdash \texttt{true} \vee Q = \texttt{true}$
4. $\vdash \texttt{false} \vee Q = Q$
5. $\vdash (P \vee Q) \vee R = P \vee (Q \vee R) = P \vee Q \vee R$
6. $\vdash P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$
7. $\vdash P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$
8. $\vdash P \wedge Q = Q \wedge P$
9. $\vdash \texttt{true} \wedge Q = Q$
10. $\vdash \texttt{false} \wedge Q = F$
11. $\vdash (P \wedge Q) \wedge R = P \wedge (Q \wedge R) = P \wedge Q \wedge R$
12. $\vdash P \Rightarrow Q = \neg P \vee Q$
13. $\vdash \neg(P \Rightarrow Q) = P \wedge \neg Q$
14. $\vdash P \Rightarrow Q \Rightarrow R = P \Rightarrow (Q \Rightarrow R) = P \wedge Q \Rightarrow R$
15. $\vdash (P = Q) = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

Theorem 2.8.13 : de Morgan

1. $\vdash \neg(P \vee Q) = \neg P \wedge \neg Q$
2. $\vdash \neg(P \wedge Q) = \neg P \vee \neg Q$

Theorem 2.8.14 : Contra Position

$\vdash P \Rightarrow Q = \neg Q \Rightarrow \neg P$

Theorem 2.8.15 : COND conversion

1. $\vdash \quad P \quad \Rightarrow \quad (P \rightarrow e_1 \mid e_2 \quad = \quad e_1)$

2. $\vdash \quad \neg P \quad \Rightarrow \quad (P \to e_1 \mid e_2 \quad = \quad e_2)$
3. $\vdash P \to e \mid e \ = \ e$
4. $\vdash f\ (P \to e_1 \mid e_2) \ = \ P \to f\ e_1 \mid f\ e_2$
5. $\vdash P \to e_1 \mid e_2 \ = \ \neg P \to e_2 \mid e_1$

Theorem 2.8.16 : COND Split

$\vdash \quad P \to Q \mid R \quad = \quad (P \Rightarrow Q) \wedge (\neg P \Rightarrow R)$

P, Q, and R have to be predicates.

Theorem 2.8.17 : Renaming bound variables

1. $\vdash (\forall i : P : Q) = (\forall i' : P[i'/i] : Q[i'/i])$
2. $\vdash (\exists i : P : Q) = (\exists i' : P[i'/i] : Q[i'/i])$

i' should not occur free in P and Q.

Theorem 2.8.18 : Negate ∀

$\vdash \neg(\forall i : P\ i : Q\ i) = (\exists i : P\ i : \neg(Q\ i))$

Theorem 2.8.19 : Negate ∃

$\vdash \neg(\exists i : P\ i : Q\ i) = (\forall i : P\ i : \neg(Q\ i))$

Theorem 2.8.20 : Nested quantifications

1. $\vdash (\forall i, j :: P\ i\ j) = (\forall i :: (\forall j :: P\ i\ j))$
2. $\vdash (\exists i, j :: P\ i\ j) = (\exists i :: (\exists j :: P\ i\ j))$

Theorem 2.8.21 : Quantification over singleton domain

1. $\vdash (\forall i : i = e : P\ i) = P\ e$
2. $\vdash (\exists i : i = e : P\ i) = P\ e$

Theorem 2.8.22 : Range Split

1. $\vdash (\forall i : P\ i : Q_1\ i \wedge Q_2\ i) = (\forall i : P\ i : Q_1\ i) \wedge (\forall i : P\ i : Q_2\ i)$
2. $\vdash (\exists i : P\ i : Q_1\ i \vee Q_2\ i) = (\exists i : P\ i : Q_1\ i) \vee (\exists i : P\ i : Q_2\ i)$

Theorem 2.8.23 : Domain split

1. $\vdash (\forall i : P\ i \vee Q\ i : R\ i) = (\forall i : P\ i : R\ i) \wedge (\forall i : Q\ i : R\ i)$
2. $\vdash (\exists i : P\ i \vee Q\ i : R\ i) = (\exists i : P\ i : R\ i) \vee (\exists i : Q\ i : R\ i)$

Theorem 2.8.24 : Quantification over empty domain

1. $\vdash (\forall i : \texttt{false} : P\ i) = \texttt{true}$
2. $\vdash (\exists i : \texttt{false} : P\ i) = \texttt{false}$

Theorem 2.8.25 : Domain Shift

1. $\vdash (\forall i : P\ i : Q\ i) = (\forall i :: P\ i \Rightarrow Q\ i)$
2. $\vdash (\forall i : P_1\ i \wedge P_2\ i : Q\ i) = (\forall i : P_1\ i : P_2\ i \Rightarrow Q\ i)$
3. $\vdash (\exists i : P\ i : Q\ i) = (\exists i :: P\ i \wedge Q\ i)$
4. $\vdash (\exists i : P_1\ i \wedge P_2\ i : Q\ i) = (\exists i : P_1\ i : P_2\ i \wedge Q\ i)$

2.9 Inference Rules of Hoare Logic

Rule 2.9.1 : Post-condition Weakening

$$\frac{\begin{array}{c}\vdash\ Q \Rightarrow Q' \\ \{*\,P\,*\}\ S\ \{*\,Q\,*\}\end{array}}{\{*\,P\,*\}\ S\ \{*\,Q'\,*\}}$$

Rule 2.9.2 : Pre-condition Strengthening

$$\frac{\begin{array}{c}\vdash\ P \Rightarrow P' \\ \{*\,P'\,*\}\ S\ \{*\,Q\,*\}\end{array}}{\{*\,P\,*\}\ S\ \{*\,Q\,*\}}$$

Rule 2.9.3 : Hoare Triple Disjunction

$$\frac{\begin{array}{c}\{*\,P_1\,*\}\ S\ \{*\,Q_1\,*\} \\ \{*\,P_2\,*\}\ S\ \{*\,Q_2\,*\}\end{array}}{\{*\,P_1 \vee P_2\,*\}\ S\ \{*\,Q_1 \vee Q_2\,*\}}$$

Rule 2.9.4 : Hoare Triple Conjunction

$$\frac{\begin{array}{c}\{*\,P_1\,*\}\ S\ \{*\,Q_1\,*\} \\ \{*\,P_2\,*\}\ S\ \{*\,Q_2\,*\}\end{array}}{\{*\,P_1 \wedge P_2\,*\}\ S\ \{*\,Q_1 \wedge Q_2\,*\}}$$

Rule 2.9.5 : SEQ

$$\frac{\begin{array}{c}\{*\,P\,*\}\ S_1\ \{*\,P'\,*\} \\ \{*\,P'\,*\}\ S_2\ \{*\,Q\,*\}\end{array}}{\{*\,P\,*\}\quad (S_1;\ S_2)\quad \{*\,Q\,*\}}$$

Rule 2.9.6 : If-Then-Else 1

$$\frac{\begin{array}{c}\{*\,P \wedge g\,*\}\ S_1\ \{*\,Q\,*\} \\ \{*\,P \wedge \neg g\,*\}\ S_2\ \{*\,Q\,*\}\end{array}}{\{*\,P\,*\}\quad \texttt{if}\ g\ \texttt{then}\ S_1\ \texttt{else}\ S_2\quad \{*\,Q\,*\}}$$

Rule 2.9.7 : If-Then-Else 2

$$\frac{\begin{array}{c}\{*\,P_1\,*\}\ S_1\ \{*\,Q\,*\} \\ \{*\,P_2\,*\}\ S_2\ \{*\,Q\,*\}\end{array}}{\{*\,(g \Rightarrow P_1) \wedge (\neg g \Rightarrow P_2)\,*\}\quad \texttt{if}\ g\ \texttt{then}\ S_1\ \texttt{else}\ S_2\quad \{*\,Q\,*\}}$$

Rule 2.9.8 : Assignment

$$\frac{-}{\{*\,Q[e/v]\,*\}\quad v := e\quad \{*\,Q\,*\}}$$

Rule 2.9.9 : Loop 1 (Partial Correctness)

$$\frac{\begin{array}{c}\vdash\ P \Rightarrow I \\ \{*\,I \wedge g\,*\}\ S\ \{*\,I\,*\} \\ \vdash I \wedge \neg g \Rightarrow Q\end{array}}{\{*\,P\,*\}\ \ \texttt{while}\ g\ \texttt{do}\ S\ \ \{*\,Q\,*\}}$$

Rule 2.9.10 : Loop 2 (Partial Correctness)

$$\frac{\begin{array}{c}\{*\,I \wedge g\,*\}\ S\ \{*\,I\,*\} \\ \vdash I \wedge \neg g \Rightarrow Q\end{array}}{\{*\,I\,*\}\ \ \texttt{while}\ g\ \texttt{do}\ S\ \ \{*\,Q\,*\}}$$

Rule 2.9.11 : Loop 3 (Total Correctness)

$$\frac{\begin{array}{c}\vdash P \Rightarrow I \\ \{*\ I \wedge g\ *\}\ \ (C := m; S)\ \ \{*\ m < C\ *\} \\ \vdash I \wedge g \Rightarrow m > 0 \\ \vdash I \wedge \neg g \Rightarrow Q \\ \{*\ I \wedge g\ *\}\ \ S\ \ \{*\ I\ *\}\end{array}}{\{*\ P\ *\}\ \ \texttt{while}\ \ g\ \texttt{do}\ S\ \ \{*\ Q\ *\}}$$

Rule 2.9.12 : Assignment Targeting an Array's Element
Treat an assignment $a[e_1] := e_2$ as an assignment:

$$a := a(e_1\ \texttt{repby}\ e_2)$$

where $a(e_1\ \texttt{repby}\ e_2)$ is defined as follows:

$$a(e_1\ \texttt{repby}\ e_2)[j]\ \ =\ \ (j = e_1) \rightarrow e_2 \mid a[j]$$

Rule 2.9.13 : Assignment Targeting an Record's Element
Treat an assignment $r.fname := e$ as an assignment:

$$r := r(fname\ \texttt{repby}\ e)$$

where `repby` is defined as follows:

$$\begin{array}{lcll} r(fname\ \texttt{repby}\ e).fname & = & e & \\ r(fname\ \texttt{repby}\ e).gname & = & r.gname & ,\text{if } fname \neq gname \end{array}$$

2.10 Esc/Java Core Logic

ESC/Java stands for *Extended Static Checker for Java.* It is a verification tool for Java. The idea is to use it as a debugger for catching common errors that are cannot be caught by Java's type checker. Example of those errors are attemp to dereference a null pointer, or access to an array beyond its range.

It works by translating a Java program to a simple imperative language called *Guarded Command Language* (GCL). It is very much like uPL. ESC/Java's GCL is slightly different than the original GCL, due to Dijkstra. Anyway, the whole point of using this indirection in ESC/Java is that GCL is a very simple language, and so is its logic. Therefore the logic can be easily implemented, and experimented with if one wishes to. The translation from Java to GCL is much more complicated, but that we can keep that freezed (avoiding experimenting with it).

Once translated to GCL, correctness criteria of a target class is converted, using Hoare logic, to verification conditions. Basically these are a bunch of predicates of the form $P \Rightarrow P'$, where P is a given pre-condition, and P' is a calculate pre-condition inferred by the logic. The class is correct if all verification conditions are valid.

To prove the verification conditions, ESC/Java uses a backend theorem prover. The whole process is automatic.

However, there is a limit in what a machine can automatically prove. Therefore, you cannot expect ESC/Java to be able to automatically prove complex specification. As said, its primary purpose is to automate the detection of simple but common errors.

Furthermore, because your class may contain loops, it may be necessary to manually annotate the loops with invariants. However you only need to put those details in your invariants that are relevant towards proving the absence of simple errors as meant above.

2.10.1 GCL

GCL only has the following commands:

1. `skip`.
2. Assignment: *var* = *expr*.
3. *cmd* ; *cmd*.
4. Throwing an exception: `raise`.
5. Try cmd_1, if it throws an exception, handles it with cmd_2: cmd_1 ! cmd_2.
6. Requiring that a predicate holds: `assert` *expr*.
7. Assuming that a predicate holds: `assume` *expr*.
8. Non-deterministically choose between cmd_1 and cmd_2: cmd_1 [] cmd_2.
9. Introducing a block with local variables: `var` $variable^+$ `in` *cmd* `end`. The variables are uninitialized. If they must be initialized, this is done explicitly in *cmd*.

2.10.2 Representing objects in GCL

Each object is assumed to have a unique natural number ID. We introduce a global variable N that keeps track of the actual number of objects that exists. So, the IDs will range from 0 to N.

We introduce a global infinite array H in GCL to represent the objects. H is a two dimensional array, indexed with field names and object IDs, such that $H[x, i]$ holds the value of the field x of the object whose ID is i.

So, Java assignment like $u.x = v.x + 1$ can now be translated to GCL assignment:

$$H[x, u] = H[x, v] + 1$$

Object creation like $u =$ `new Point`() is translated to:

$$\begin{array}{l} u = N \ ; \\ H[x, u] = 0 \ ; \\ H[y, u] = 0 \ ; \\ N = N{+}1 \end{array}$$

2.10.3 ESC/Java's GCL logic

The state of a GCL program can be thought as being labelled by either normal, exceptional, or error flags. The command `raise` is the only one that can switch the state's flag to exceptional. It does nothing to the state itself. If the control proceeds to an exeception handler, thus in the $C!D$ structure, the flag is set back to normal.

Violating `assert` will cause the flag to be set to error. This status will stick (once flagged as error, it will remain so).

We will extend the Hoare triple such that the post-condition now consists of three parts:

$$\{* \ P \ *\} \ \ S \ \ \{* \ N, X, W \ *\}$$

N, X, W are predicates. N is the post-condition that should hold if S terminates in a normal state; X if it terminates in an execeptional state; and W if it terminates in an error state.

Rules for calculating wlp:

1. wlp `skip` $(N, -, -) \quad = \quad N$

2. $\mathsf{wlp}\ (x = e)\ (N, -, -) \quad = \quad N[e/x]$
3. $\mathsf{wlp}\ \mathtt{raise}\ (-, X, -) \quad = \quad X$
4. $\mathsf{wlp}\ (\mathtt{assert}\ P)\ (N, -, X) \quad = \quad (P \wedge N) \vee (\neg P \wedge X)$
5. $\mathsf{wlp}\ (\mathtt{assume}\ P)\ (N, -, -) \quad = \quad P \Rightarrow N$
6. $\mathsf{wlp}\ (C[]D)\ (N, X, W) \quad = \quad (\mathsf{wlp}\ C\ (N, X, W))\ \wedge\ (\mathsf{wlp}\ D\ (N, X, W))$
7. $\mathsf{wlp}\ (C; D)\ (N, X, W) \quad = \quad \mathsf{wlp}\ C\ (\mathsf{wlp}\ D\ (N, X, W),\ X,\ W)$
8. $\mathsf{wlp}\ (C!D)\ (N, X, W) \quad = \quad \mathsf{wlp}\ C\ (N,\ \mathsf{wlp}\ D\ (N, X, W),\ W)$
9. For simplicity we assume that x is a fresh variable. Else rename it to make it fresh.

   $\mathsf{wlp}\ (\mathtt{var}\ x\ \mathtt{in}\ C\ \mathtt{end})\ Q \quad = \quad (\forall x :: \mathsf{wlp}\ C\ (N, X, W))$

2.10.4 How ESC/Java handles loops

As you notice (our) GCL does not have a loop. ESC/Java removes loops by 'unrolling' them some fixed number of times. Let this be a loop, annotated with an invariant:

```
while g (inv I) do S
```

If no invariant is annotated, then `true` is used. This is unrolled to:

```
assert I ;
 if g then {S; assert I; assume ¬g} else skip
```

There is still the if-then-else structure that we also do not have in GCL, but translating it to GCL is quite straight forward.

After the above unrolling, then we simply expose it to GCL logic. This effectively verify the following:

1. That I is established as we enter the loop.
2. That the first iteration re-establishes I.
3. If the loop terminates in 0 or 1 iteration, the rest of the program will be verified with no information loss.

If the loop actually iterates more than once, then it assumes falsity, thus effectively becomes blind to what happens after that.

The above translation is indeed incomplete. It will still find bugs which can occur in 0 or 1 iteration; but not those bugs which are only reachable after further iterations.

You can still opt to unroll the loops further, e.g. up to 10 times. This increases your coverage, but the bottom line is that it is still incomplete. However, as a debugger the author claims that it is quite effective (it finds many bugs).

Chapter 3

Model Checking

3.1 Automata

Figure 3.1 shows some examples of *finite state automata*. They are often used to (abstractly) model programs. The nodes in an automaton are the automaton's states, and the arrows depict how the automaton can go from one state to another. They are also called *transitions* or *actions*.

The short arrow pointing to s (on the left) is used to denote that s is the automaton starting/initial state.

An *execution* of an automaton is a path through the automaton, starting with an initial state. An execution can be finite, or infinite. Sometimes we only want to consider finite executions, sometimes only infinite executions, depending on your modeling purpose. If the automaton has a concept of accepting states, we may only want to consider executions that end in an accepting states. And so on.

State properties, e.g. that in all states of an automaton the value of x should be 0, can be verified simply by checking the properties on each state. Afterall, we only have finitely many of them. Verifying properties on executions is different, because a finite-state automaton can generate infinitely many executions (as the automa Figure 3.1). So, simply enumerating the executions will not work. Furthermore, some executions may be infinite. We will see more on verification later.

There are many kinds of automata, depending of your modeling purpose. Some automata may allow multiple initial states, for example.

Some automata have labels on the arrows, e.g. as the middle and right automata. These labels can mean different things, again depending on your modeling purpose. In our examples above they represent the name or identity of the actions. So, in both automata we have two possible actions: a and b.

The automaton in the middle is *deterministic*, because every action (either a or b) uniquely takes some state x to a destination state y. The automaton on the right is *non-deterministic*, because executing the action a on the state s takes the automaton to either the state s again or the state t.

A state with no successor is a terminal state: no further execution is possible from such a state. However, some automata also allow some states to be marked as e.g. *accepting states*. An accepting state is often denoted with double line, as the state t in the middle automaton. An accepting state is often used to model that a certain important task has been acomplished; again this depends on your modeling purpose.

A real program P operates on variables. The program's *concrete state* is determined by the values of its variables. When we models the program with an automaton M, we basically map P's concrete states to M's states. This mapping can be one-to-one, but it does not have to.

If the mapping is one-to-one, we can describe it with a bijective function V that maps M's states to P's concrete state. So, given a state s of M, $V(s)$ will give us P's concrete state that corresponds to s, and coversely, if z is a concrete state, $V^{-1}(z)$ returns M's abstract state that

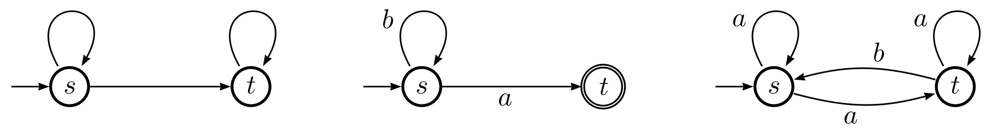


Figure 3.1: Finite state automata

represents z.

We can also choose to model more abstractly, by only specifying a fixed set of properties over P's concrete states. We can query which properties hold or do not hold on a given state in the model M, but we will have no information on properties which were not included in the model. For describing this kind of models we use a variant of automata called *Kripke structure*.

Definition 3.1.1 : Kripke Structure
A Kripke structure K is a finite state automaton, described by a tuple $(S, s_0, R, \mathit{Prop}, V)$ where:

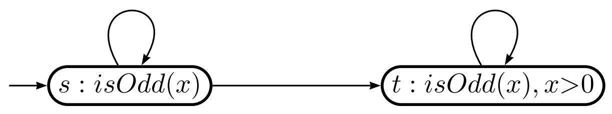
- S is the set of states, with s_0 as the initial state.
- $R : S \to \mathit{Pow}(S)$ describes the arrows. If s is a state in K, the $R(s)$ gives the set of all possible next-states.

  The notation $\mathit{Pow}(S)$ denotes the set of all subsets of S. It is sometimes also denoted as 2^S.
- Prop is a set of atomic propositions. Each abstractly describes some property on the program being modelled by K.
- $V : S \to \mathit{Pow}(\mathit{Prop})$ is called a *labelling function*. If s is a state in K, $V(s)$ is the set of all propositions that hold in s.

  Furthermore, we take the convention that if $a \notin V(s)$ then a does *not* hold in s.

□

Example:



We have above two states, named s and t. We have two propositions, $\mathit{isOdd}(x)$ and $x{>}0$. We can see above that $\mathit{isOdd}(x)$ holds in s, but $x{>}0$ does not, whereas in state t both propositions hold.

Formally, the R of the above Kripke structure is:

$$\begin{array}{lll} s & \mapsto & \{s, t\} \\ t & \mapsto & \{t\} \end{array}$$

And the V:

$$\begin{array}{lll} s & \mapsto & \{\mathit{isOdd}(x)\} \\ t & \mapsto & \{\mathit{isOdd}(x), x{>}0\} \end{array}$$

A Kripke structure is *non-deterministic*, if there is state that has multiple next-states. The above example is non-deterministic.

The arrows in a Kripke structure are not labelled, and the structure has no concept of acceptance state either. States with no successor implicitly model terminal states.

The labelling with the propositions is assumed to be semantically consistent. That is:

- If $a, b \in V(s)$, then a and b shold not be semantically contradictory. In other words, $a \wedge b$ should still be satisfiable.
- If $a \in V(s)$ and $b \in A$, but $b \notin V(s)$, then a and $\neg b$ should not be semantically contradictory. In other words, $a \wedge \neg b$ should be satisfiable.

3.2 Some Basic Operations on Automata

In this section we will assume the following concept of automaton. An automaton M is a tuple (S, I, Σ, R) where:

- S is the set of states. I is a non-empty subset of S, specifying M's possible initial states.
- Σ are the set of actions of M. Every arrow in M is labelled with one action.
- $R : S \to \Sigma \to Pow(S)$ describes the arrows. If s is a state, and a is an action, $R(s, a)$ is the set of possible next-states if we execution the action a on the state s.

An *execution* of M is a path through M that starts in an initial state. A path is a sequence of connected arrows in the automaton. It can be finite or infinite. The i-th action of the execution is the action that label its i-th arrow. The i-th state in the execution is the starting state of the i-th arrow in the sequence. If the execution is finite, its last state is the end-state of its last arrow.

If τ is an execution, we will use the notation τ_i or $state(\tau, i)$ to denote it's i-th state, and τ^i or $act(\tau, i)$ to denote it's i-th action.

Sometimes we are more interested in the sequence of actions taken along an execution. We will call this sequence *sentence*; sometimes it is also called *trace*. So, if τ is an execution, its induced sentence is the sequence $\tau^0, \tau^1, ...$ We say that s is a sentence of M if there is an execution of M that induces it.

Sometimes we are more interested in the sequence of states along the execution. Rather than inventing yet another name, let us just call this sequence of states '*execution*'.

3.2.1 Intersection of automata

We can define the intersection of two automata M and N simply by taking the intersection of the set of arrows of both automata. Whether this definition is good depends of course on our purpose: what do we want to do with it?

For our purpose later, we will take a bit more complicated concept of intersection. The idea is that $M \cap N$ should be an automaton whose sentences are exactly those sentences allowed by both M and N.

The idea is to construct $M \cap N$ by executing both M and N. We will keep track where we are (in which states we are) in M and in B; and we will do a transition labelled with an action a, only if this action is possible in both M and N.

Definition 3.2.1 : INTERSECTION
Let $M = (S_1, I_1, \Sigma_1, R_1)$ and $N = (S_2, I_2, \Sigma_2, R_2)$. We define $M \cap N = (S, I, \Sigma, R)$ where:

- $S \;=\; S_1 \times S_2$. So, the states of M are pairs (s, t). This is how we keep track of the states of M and N; s is where we are in M; and t is where we are in N.
- $I \;=\; \{(s_0, t_0) \mid s_0 \in I_1 \wedge t_0 \in I_2\}$
- $\Sigma \;=\; \Sigma_1 \cap \Sigma_2$
- R is such that $(s', t') \in R((s, t), a)$ if and only if $s' \in R_1(s, a)$ and $t' \in R_2(t, a)$.

□

$M \cap N$ is also called the automaton obtained by 'lock-step execution' of M and N.

3.2.2 Interleaving of automata

Let M and N be two automata with no common action. The interleaving of them, denoted by $M||N$ is the automaton whose execution is obtained by interleaving the arrows of M and N.

$M||N$ is also called the product automaton of M and N.

$M||N$ represents a composite system, obtained by executing M and N in parallel. We do assume here that they operate on their own private states, and that they can do their actions independently (no need to synchronize). The state of $M||N$ is formed by tupling the states of M and N.

Definition 3.2.2 : INTERLEAVING
Let $M = (S_1, I_1, \Sigma_1, R_1)$ and $N = (S_2, I_2, \Sigma_2, R_2)$, such that they have no common action. We define $M||N = (S, I, \Sigma, R)$ where:

- $S = S_1 \times S_2$. So, each state of $M||N$ is a pair (s, t). Think this as a joint state, where the s component tells us what the state of M is within the composite $M||N$, and similarly the t component tells us the state of N.
- $I \;=\; \{(s,t) \mid (s,t) \in I_1 \times I_2\}$
- $\Sigma \;=\; \Sigma_1 \cup \Sigma_2$
- R is such that $R((s,t),a) \;=\; \{(s',t) \mid s' \in R_1(s,a)\} \;\cup\; \{(s,t') \mid t' \in R_2(t,a)\}$

□

Projecting the sentences of $M||N$ to Σ_1 will give us all the sentences of M; projecting them to Σ_2 gives us the sentences of N.

Synchronized actions

A variations of the above modeling of parallel composition is when M and N share some actions. We first need to decide how such a common action is executed in a parallel execution of M and N. One possibility, which is quite common in varios modelling approaches, is to consider a common action as an action that must be executed *together*. So if a is a common action, the composite $M||N$ can make a transition:

$$(s,t) \quad \xrightarrow{a} \quad (s',t')$$

if and only if M can go from s to s', with the action a, *and* if N too can go from t to t' with the action a. This is also called *synchronous* execution of a.

You need to adapt the definition of interleaving $M||N$ accordingly.

When the automata operate on a common state space

Another variation is if M and N actually operate on the same set of states, rather than on their respective private states. So, $S_1 = S_2$. In this case we should take $S = S_1$ as $M||N$'s set of states; thus not $S_1 \times S_2$. The arrows of $M||N$ are then either the arrows of M or the arrows of N, labelled by non-common actions, or synchronous transitions by both M and N over their common actions.

3.3 Temporal Properties

A *temporal property* is a 'timing' dependent property. Usually, relative timing is meant, as opposed to real time property. Abstractly it can be characterized as a property over the executions of an automaton. Examples of such a property is:

- Whenever R receives a value through a channel c, the value it receives is never 0.

- $S||R$ won't deadlock.

In contrast, Hoare triple only specifies a relation between the initial and end-state of an execution. In that respect, Hoare triple is just a special case of temporal properties. However, most temporal properties cannot be expressed with Hoare triples.

One way to specify temporal properties is by using *Linear Temporal Logic* (LTL). It provides a number of operators to express temporal properties, e.g. $p \to \Diamond q$ means that if p holds initially, then eventually q will hold.

An LTL *formula* is an expression with the syntax below; ϕ, ψ range over LTL formulas, and p ranges over atomic propositions.

$$\begin{array}{lll} \phi & ::= & p, \text{ where } p \text{ is an atomic proposition} \\ & ::= & \neg\phi \ \mid\ \phi \wedge \psi \ \mid\ \mathbf{X}\ \phi \ \mid\ \phi\ \mathbf{U}\ \psi \end{array} \tag{3.1}$$

Intuitively, $\mathbf{X}\ \phi$ means that ϕ holds on the next state. Whereas $\phi\ \mathbf{U}\ \psi$ means that either ψ holds now, or in the future. If it holds in the future, then from now until the point just before it holds, ϕ holds.

Notice that the syntax allows you to nest temporal operators; e.g. you can write: $p\ \mathbf{U}\ (q\ \mathbf{U}\ r)$.

Additionally, we will also introduce the following derived operators:

- Disjunction and implication:

$$\begin{array}{lll} \phi \vee \psi & = & \neg(\neg\phi \wedge \neg\psi) \\ \phi \to \psi & = & \neg\phi \vee \psi \end{array}$$

- Eventually ϕ:

$$\Diamond\phi \quad = \quad \texttt{true}\ \mathbf{U}\ \phi$$

- Always ϕ:

$$\Box\phi \quad = \quad \neg\Diamond\neg\phi$$

- ϕ weak-until ψ; which means either ϕ holds forever, or we switch over to ψ (but we don't have to switch over):

$$\phi\ \mathbf{W}\ \psi \quad = \quad (\Box\phi) \vee (\phi\ \mathbf{U}\ \psi)$$

  The operator is also called 'unless'.

- ϕ releases ψ; which means either ψ holds forever, or at some point it is released by $\phi \wedge \psi$:

$$\phi\ \mathbf{R}\ \psi \quad = \quad \phi\ \mathbf{W}\ (\phi \wedge \psi)$$

Some examples of properties you can express with LTL:

- $\Box(p \to \Diamond q)$: whenever p holds, then eventually q will hold.
- $p\ \mathbf{U}\ (q\ \mathbf{U}\ r)$: we start with a duration where p holds constantly, followed immediately by a duration where q holds constantly, which will be ended by a point where r holds. Each duration above can be empty.
- $\Diamond\Box p$: eventually we will come to p, afterwhich we will remain in p. So, this descrives a system that stabilizes towards p.

To simplify our discussion, we will only define the semantics of LTL operators with respect to infinite executions. If the given automaton contains a terminal state, and we can add a self-looping arrow on that state, and thus making all of its executions infinite. We can then add a label that only holds on the terminal state, and thus can recover the ability to specify properties on the terminal state by expressing it through this unique label.

We will model the target system with a Kripke structure $M = (S, s_0, R, Prop, V)$ be a Kripke stucture (with no terminal state). If the system has concurrent components, and you have modelled each component with its own automaton, we assume that M is the combined automaton of all those components, e.g. using the interleaving operation from Section 3.2.

An *execution* of a Kripke structure is as defined generically in Section 3.2, except that the arrows of a Kripke structure has no label. It the induces a sequence of propositions that hold along the execution; we will call this sequence *abstract execution*, or just *execution* if it is obvious from the context which one is meant. To be precise, if τ is an execution, its induced abstract execution is $V(\tau_0), V(\tau_1), ...$

3.3.1 Valid Property

We will write $M \models \phi$ to denote that the property ϕ is a valid property of the Kripke structure M. This means that it is valid on all abstract executions of M.

Semantically, an LTL property is a predicate over (abstract) executions. If Π is an abstract execution of M, we write $\Pi \models \phi$ to mean that ϕ is a valid property on Π.

Note that since an execution is infinite, its sufix is also infinite. We write $\Pi, i \models \phi$ to mean that ϕ is a valid property of the sufix of Π, starting from its i-th element. So:

$$\Pi \models \phi \quad = \quad \Pi, 0 \models \phi$$

Let $\Pi(i)$ denotes the i-th element of the sequence Π. Note that this is a set of proposition (from M's $Prop$).

Now we can define the meaning of our LTL formulas:

1. If p is an atomic proposition (from $Prop$), $\Pi, i \models p$ means that p holds on Π's i-th state. So:

$$\Pi, i \models p \quad = \quad p \in V(\Pi(i))$$

2. $\neg\phi$ is valid on Π, i, if its negation is not valid. So:

$$\Pi, i \models \neg\phi \quad = \quad \text{not } (\Pi, i \models \phi)$$

E.g. in the case of atomic proposition p, then:

$$\Pi, i \models \neg p \quad = \quad p \notin V(\Pi(i))$$

3. $\phi \wedge \psi$ is valid on Π, i if each is individually valid. So:

$$\Pi, i \models \phi \wedge \psi \quad = \quad (\Pi, i \models \phi) \text{ and } (\Pi, i \models \psi)$$

4. $\mathbf{X}\ \phi$ is valid on Π, i if ϕ holds from the next state:

$$\Pi, i \models \mathbf{X}\ \phi \quad = \quad \Pi, i{+}1 \models \phi$$

5. $\phi\ \mathbf{U}\ \psi$ is valid on Π, i if at some time in the future of i, ψ holds, and in the mean time ϕ holds:

$$\begin{aligned}\Pi, i \models \phi\ \mathbf{U}\ \psi \quad = \quad & \text{there is a } j \geq i, \text{ such that } \Pi, j \models \psi \\ & \textbf{and } (\forall h : i {\leq} h {<} j : \Pi, h \models \phi)\end{aligned}$$

3.4 Buchi Automaton

Some automata have *acceptance states*. We can define an execution of M as *accepting* if it ends in an acceptance state of M. A sentence of M is said to be *accepted* by M if it is induced by an accepting execution. So, while M can produce lots of sentences, not all are considered as 'accepted sentences'. Let $Lang(M)$, 'the *language* of M', denote the set of all accepted sentences of M.

The above is the standard definition of acceptance. Note that only finite sentences can thus be accepted (because it must *ends* in some state). Because in LTL we are dealing with inifinite sentences we will tweak the concept of 'acceptance'. An automaton with this tweaked acceptance is called *Buchi Automaton*.

Definition 3.4.1 : Buchi Automaton
Is a finite state automaton, described by a tuple $M = (S, I, F, R, \Sigma)$. Most of the components are the same as before, except that now we have F. There is no labelling function.

- S is the set of states. I is a non-empty subset of S, specifying M's possible initial states.
- F is a subset of S, this is the set of M's acceptance states.
- Σ is the 'alphabet' of M. It is the set of labels of M's arrows. Every arrow in M is labelled with one symbol from Σ.
- $R : S \to \Sigma \to Pow(S)$ describes the arrows. If s is a state, and a is a symbol, $R(s, a)$ is the set of possible next-states we arrive at by 'cosuming' the symbol a on the state s. We can only move to a next state by consuming a symbol.

  Note that such an R allows M to be non-deterministic.

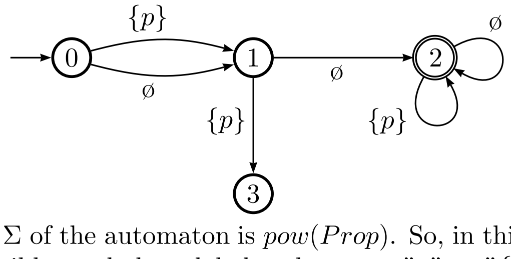
□

A Buchi automaton M only accepts infinite sentences. A infinite execution τ of M is accepting if it passes through F infinitely many times. Since F contains only finitely many states, this is the same as saying that τ will pass some acceptance state infinitely many times (but it does *not* have to pass *every* member of F infinitely many times).

An infinite sentence is accepted if it is induced by an accepting execution. $Lang(M)$ is defined as before.

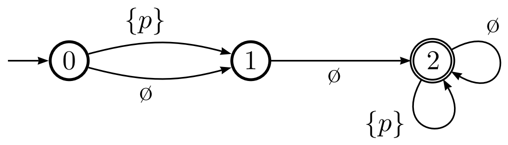
We can use a Buchi automaton to represent an LTL formula. Let ϕ be an LTL formula over *Prop* as its set of atomic propositions. Now, you have seen that semantically an LTL formula is defined as a predicate over abstract executions. An abstract execution is an infinite sequence over *pow*(*Prop*). The idea is to construct a Buchi automaton M_ϕ that accepts all possible infinite sentences over *pow*(*Prop*) on which ϕ holds. This automaton would then fully describe ϕ. This automaton a bit wierd because its alphabet Σ must be *pow*(*Prop*) (and *not Prop*).

For example, the Buchi automaton below fully describes the abstract executions of the formula $\mathbf{X}\ \neg p$, over $Prop = \{p\}$:



Note that the alphabet Σ of the automaton is *pow*(*Prop*). So, in this case $\Sigma = \{\emptyset, \{p\}\}$, which means that we have to possible symbols to label each arrow: "ø" or "{p}".

The automaton can be a bit simpler though. In state-2 we describe where the automaton would go if it consumes each of possible symbol. But because the transition to state-3 will never lead to an accepting execution, we can just as well remove it. So, we obtain this:



If we now take a larger $Prop = \{p, q\}$, the automaton will look quite verbose. E.g. we would now have four arrows from state-0 to state-2, labelled by each of the subset of $\{p, q\}$. The same with the loop from state-2 to state-2. To simplify the drawing, we will use the following graphical shorthand:

- $s \xrightarrow{*} t$

  Intuitively, this means that we can go from s to t by consuming any symbol.

  Technically, it represents a bunch of arrows from state s to t: for each subset A of $Prop$, we implicitly have the arrow $s \xrightarrow{a} t$.

- $s \xrightarrow{p\in} t$

  We can go from s to t if the proposition p holds.

  Technically, it represents a bunch of arrows from state s to t: for each subset A of $Prop$ such that $p \in A$, we implicitly have the arrow $s \xrightarrow{a} t$.

- $s \xrightarrow{p\notin} t$

  We can go from s to t if the proposition p does not hold.

  Technically, it represents a bunch of arrows from state s to t: for each subset A of $Prop$ such that $p \notin A$, we implicitly have the arrow $s \xrightarrow{a} t$.

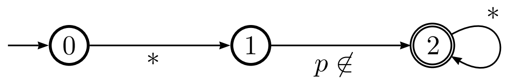
- $s \xrightarrow{C_1, C_2, ...} t$

  We can go from s to t if all the conditions hold.

  Technically, it represents a bunch of arrows from state s to t: for each subset A of $Prop$ such that A satisfies all the conditions $C_1, C_2, ...$, we implicitly have the arrow $s \xrightarrow{a} t$.

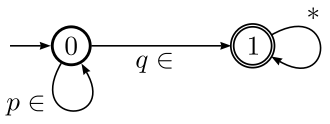
  E.g. we can thus write $s \xrightarrow{p\in, q\notin} t$

So now we can draw the previos automaton less verbosely, and moreover the drawing is now independent of the choice of $Prop$:

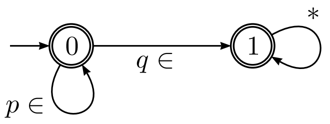


Here are few more examples:

- A Buchi automaton that describes the formula p **U** q:



- A Buchi automaton that describes the formula p **W** q; notice that we have two accepting states:



- A Buchi automaton that describes the formula $\Diamond p$:

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[loop below, "*"] \arrow[rr, "p \in"'] & & 1 \arrow[loop right, "*"]
\end{tikzcd}$$

- A Buchi automaton that describes the formula $\Box p$:

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[loop right, "p \in"]
\end{tikzcd}$$

3.4.1 Non-determinism in Buchi Automata

Notice that the definition of Buchi Automata (Def. 3.4.1) allows it to be non-deterministic. You may wonder whether a non-deterministic Buchi automaton can be converted to a deterministic one. For ordinary state automata this is indeed the case, but not for Buchi. An example where it is possible is the above automaton for $\Diamond p$; it is non-deterministic. But we can equivalently describe it by:

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[loop below, "p \notin"] \arrow[rr, "p \in"'] & & 1 \arrow[loop right, "*"]
\end{tikzcd}$$

However take a look at the automaton below, describing $\Diamond\Box p$. There is no deterministic Buchi that can describe the formula.

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[loop below, "*"] \arrow[rr, "p \in"] & & 1 \arrow[loop right, "p \in"]
\end{tikzcd}$$

The formula requires that p will eventually stabilize. However, before that can happen the prefix can be anything. The only deterministic sub-automaton that can describe such a prefix is the arrow-* we see above; but then the needed outgoing arrow $p \in$ will not be disjoint with this $*$, and thus it cannot be deterministic.

So, in Buchi automata non-determinism really adds expressiveness.

3.4.2 Generalized Buchi Automata

To represents some formulas, in particular conjunctions, it is convenient to use a *generalized Buchi automaton.* It is a Buchi automaton with multiple sets of accepting states. Note that we are talking about set of sets here. An ordinary Buchi automaton may have one, or multiple, accepting states. They are grouped in a single *set* of accepting states. A generalized Buchi has multiple of such sets.

Definition 3.4.2 : GENERALIZED BUCHI AUTOMATON (GBA)
Is a finite state automaton, described by a tuple $M = (S, I, \mathcal{F}, R, \Sigma)$. All components except $\mathcal{F}$ have the same meaning as in the ordinary Buchi automaton.

$\mathcal{F}$ is a set of sets. Each member F of $\mathcal{F}$ is a set of accetance states.
□

An execution is accepting by the GBA M if it passes through *each* member of $\mathcal{F}$ infinitely many times.

For example a GBA describing the conjunction $(\Box\Diamond p) \wedge (\Box\Diamond q)$ is:

$$\begin{tikzcd}
 & {} \arrow[d] & \\
1 \arrow[r, bend right, "*"'] & 0 \arrow[l, bend right, "p \in"'] \arrow[r, bend left, "q \in"] \arrow[loop below, "*"] & 2 \arrow[l, bend left, "*"]
\end{tikzcd}$$

The $\mathcal{F}$ is $\{\{1\},\{2\}\}$. So, we must visit state-1 infinitely many times, *and* also visit state-2 infinitely many times. Note that this is different that saying, as in the ordinary Buchi automaton, that $\{1,2\}$ is our set of accepting states, for which passing state-1 infinitely many times, but ignoring state-2, will do.

Every generalized Buchi automaton can be converted into an equivalent ordinary Buchi automaton (that is, they accept the same language).

3.4.3 Constructing Buchi Automaton

Every LTL formula can be represented by a (generalized) Buchi automaton. There are algorithms to systematically construct such an automaton. We will not go into this.

3.5 LTL Model Checking

Let M be a Kripke structure. We want to verify whether $M \models \phi$. We can do this by first constructing the (generalized) Buchi automaton that represents the negation of ϕ; let's call this automaton $B_{\neg\phi}$. The intersection of these two automata produces sentences which can be produces by both M and $B_{\neg\phi}$. If one of these sentences, say σ, is also an accepted sentece for $B_{\neg\phi}$, then it follows that this σ represents an execution of M that satisfies $\neg\phi$, and thus violates ϕ. Therefore if such a σ can be found, then ϕ is not valid. In fact, we have this relation:

Theorem 3.5.1 :

$$M \models \phi \text{ if and only if } Lang(M \cap B_{\neg\phi}) = \emptyset$$

□

The good news is, intersection can be calculated —see Definition 3.2.1. Checking if the language of a Buchi automaton is empty can also be done in finite time. So, we can do verification, despite the infinite number of executions that M may generate!

Before we go into the algorithm for the latter, there is one technical detail we have to deal with. We cannot directly do the intersection $M \cap B_{\neg\phi}$ because they are two different kinds of automata. There is a slight mismatch in their concepts. In a Kripke structure, propositions label the states. Whereas in a Buchi automaton they label the arrows. The make them match, we will transform the Kripke structure M to a Buchi automaton, without changing its meaning of course.

So, if $M = (S, s_0, R, Prop, V)$ its corresponding Buchi automaton is $Buchi(M) = (S', I, F, R', \Sigma)$ where:

- $S' = S \cup \{z_0\}$, where z_0 is a new dummy innitial state.
- $I = \{z_0\}$.
- We want to consider all possible executions of M. So, it does not have any constraint on which sentences it 'accepts'. Therefore we set F to be the entire S'. In other words, every state is an acceptance state.
- $\Sigma = pow(Prop)$.
- We add a new arrow from z_0 to the actual innitial state s_0, labelled by $V(s_0)$. So:

$$R'(z_0, A) = \begin{cases} \{s_0\} & \text{, if } A = V(s_0) \\ \emptyset & \text{otherwise} \end{cases}$$

We keep the arrows in R, each will be labelled with the label of its destination. So;

$$R'(s, A) = \{t \mid t \in R(s) \text{ and } A{=}V(t)\}$$

Note that M and $Buchi(M)$ generate the same set of sentences.

Now we can construct $C = Buchi(M) \cap B_{\neg\phi}$, as described in Definition 3.2.1. This results in a new Buchi automaton. Because $Buchi(M)$ puts no constraint on its acceptance states, the acceptance states of C is those states (s,t) where t is an accepting state of $B_{\neg\phi}$.

Now the only thing left to do is to check whether $Lang(C)$ is empty; the is explained in the next section.

3.5.1 Emptiness of Buchi Automaton

Consider your Buchi automaton C. It may generate infinitely many sentences. So we cannot enumerate them to check if there is one that would be accepted by C. However, the acceptance criterion of a Buchi automaton is rather unique. Combined with the fact that, like all the other automata we discuss here, C has a finite number of states, we can conclude the following:

Theorem 3.5.2 :
The language of a Buchi automaton is *not* empty if and only if it has *one* reachable accepting state, that cycles to itself.
□

Good! This comes thus down in finding cycles in your automaton, which is a solvable problem. Notice that an automaton is also a finite and directed *graph*.

A directed graph has nodes, connected by arrows. We can describe it with a tuple $(N, next)$ where N is its set of nodes and $next : N \rightarrow pow(N)$ describes the arrows. For every node u, $v \in next(u)$ means that there is an arrow from u to v. So, $next(u)$ is just the set of all 'next'-nodes of u.

There are algorithms to calculate the set $\mathcal{C}$ of strongly connected components (SCC) of a graph. An SCC is a subgraph such that any two nodes u, v in the SCC are reachable from each other. It follows that if u is in an SCC, then there is a cycle from u to itself. So, all we need to do then is to go through this set $\mathcal{C}$ to find one SCC that contains an accepting state, which is furthermore reachable from an innitial states.

That was one way to do it. The model checker SPIN uses a different algorithm, namely depth first search (DFS). DFS is a generic algorithm on a graph. It traverses a given graph from a given node, that acts as the root. It follows the arrows as deep as possible, and then backtracks. It remembers those nodes it has visited, so that it does not visit a node twice (or else, it will not terminate). The algorithm is shown below.

To make it look simpler, in the algorithm `visited` is used as a *global* variable of type set, which is initialized to empty. The algorithm works on a graph `G` = (`N`, `next`), which is also kept as a global variable.

```
DFS(path:[Node], u:Node) {

   if (u ∈ visited) return ;

   visited.add(u) ;

   for each (s ∈  G.next(u)) DFS(path++[u],s) ;
}
```

So, if we innitialize `visited` to empty, then we call `DFS`([], `root`), where `root` is some node in `G`. After `DFS` returns, the set `visited` will contain all the nodes of `G` which are reachable from `root`, through a path of length 0 or more.

Notice that each recursive calls `DFS(p, u)` maintains the invariant that `u` is reachable from `root` through the path `p`.

The time complexity of DFS is $O(R+E)$, where R and E are the number of nodes respectively edges in G which are reachable from the `root`. This is because DFS will traverse the entire

```
// top-level call:
MC() {
   visited = ∅ ;
   for each (root ∈ G.initialStates)
      if (root ∉ visited) DFS([],root) ;
}

DFS(path,u) {

   if (u ∈ visited) return ;

   visited.add(u) ;

   for each (s ∈ G.next(u)) {
      if (u ∈ G.acceptingStates) {
         visited2 = ∅ ;
         DFScheckCycle(path++[u],u,s) ;
      }
      DFS1(path++[u],s) ;
   }
}
```

Figure 3.2: SPIN's nested DFS model-checking algorithm.

subgraph of G which are reachable from `root`, where every reachable node and edge is processed just once.

We can easily turn the algorithm to an algorithm that checks if `root` has a path that cycles back to itself:

```
DFScheckCycle(path,root,u) {

   if (u===root) { // cycle found!
      throw CycleFound(path++[u]) ;
   }

   if (u ∈ visited2) return ;

   visited2.add(u) ;

   for each (s ∈ G.next(u)) DFScheckCycle(path++[u],root,s) ;
}
```

To make it simple we let the algorithm breaks as soon as it finds a cycle (by throwing an exception). The cycle found is reported inside the thrown exception.

By nesting the two DFSs above we have our model checing algorithm, which shown in Figure 3.2. Basically DFS1 traverses the given Buchi automaton in the depth first order, starting from its innitial state z_0. If DFS1 encounters an accepting state t, this state is therefore reachable from the innitial state. It then calls DFS2 to check if we also have a cycle from t, going back to t. If such a cycle is found, we have a violation on the checked property. DFS2 will break the entire process (including DFS1), and report the cycle, prefixed with the path from z_0 to t. This reported sequence is essentially an execution demonstrating where the checked property was violated. With a proper tool you can replay and inspect this execution to figure out what the source of the problem is.

If the Buchi automaton has more than one innitial state, then at the top level we call $DFS1$ on each of the innitial state, but you don't need to reset `visited` in between. See the program `MC` in Figure 3.2.

The (upper bound) time complexity of the nested DFS algorithm is $O(|M| * 2^{|\phi|})$ where $|M|$ is the size of your target the Krikpe structure, and $|\phi$ is the size of the LTL formula you are verifying.

3.5.2 Doing it on the fly

When applied directly, the algorithm requires you to have the Kripke structure representing your program. Due to its abstraction, it can potentially be much smaller than the full state space of the program. However, to construct it it usually means that you need to first explore your program's full state space, and then we can figure out how to reduce it to some Kripke structure. So, rather than going through this we can just as well apply the algorithm directly on the program's state space.

If we simulate a program P, and explore all its possible actions, then we can construct the finite state automaton that fully captures the program's behavior. This finite state automaton is often called P's *state stapce*. Each state of this automaton would contain full information on the values of the program variables on that state. We no longer need the labelling function V that we have in Kripke structure, because with such full information we can determine if any given proposition holds on a given state. Now that you have an automaton, the algorithm works for the rest exactly the same.

Note that even a small program can generate a huge automaton. In fact, this is almost always so. An important advantage of the above DFS-based algorithm is that it allows you to model check, as at the same time you construct the automaton of your program P. This is called lazy model checking, or on the fly model checking. You can expect that in the beginning your program will contain errors. If an error can already be found after just a few steps, to first generate P's full state space (which may take a few minutes to perhaps hours, depending on your program) would be a waste time. So here, you really take the benefit of on the fly model checking. Of course if your program is correct, we cannot infer that without first exploring its entire state space.

As a final note, I keep using the term 'program' here. The term 'model' is perhaps more appropriate. You don't usually use a model checker on e.g. a real Java program. The state space it generates would simply be too large to be held in our computers. You can however construct a model of the program, either by hand, or you try to infer that automatically. At the expense of losing some details, a model is smaller, and can be feasibly handled by a model checker. A model can be described by a programming language. In principle, any language will do, but a model checking tool can usually only accept models written in some special modeling language. SPIN for example, accepts Promela.

3.6 References

- C. Baier and J. P. Katoen. *Principles of Model Checking.* MIT Press, New York, May 2008

Chapter 4

CTL Model Checking

4.1 CTL

Computation Tree Logic(CTL) is another popular class of temporal logic. It shares some similarity with LTL, but is not entirely the same. For example, navigations properties on a GUI or a web site can be better expressed with CTL than LTL.

We will again interpret CTL on Kripke structures. Remember that a Kripe structure K is always defined with respect to a set *Prop* of atomic propositions.

A CTL formula is not interpreted on execution sequences, but rather on an execution tree. An execution 'sequence' represents 'the' current execution, whereas an execution tree rooted in a certain state s represents all possible executions that can branch out from s.

In CTL it is called 'computation tree' rather than 'execution tree'.

Let K be a Krike structure. A computation tree is a tree such that every branch in the tree is an arrow in K, and it is maximal (it keep expanding until it hits terminal states). Such a tree can be infinitely deep. As with LTL, to simplify the discussion we will assume that K has no terminal state; so we will only have infinite computation trees.

The computation tree that starts in K's innitial state is of course a bit special, since this represents K's all possible full executions.

The syntax of CTL:

$$\begin{array}{lll} \phi & ::= & p, \quad \text{where } p \text{ is an atomic proposition} \\ & & \mid \neg\phi \mid \phi \wedge \psi \\ & & \mid \mathbf{EX}\ \phi \mid \mathbf{E}(\phi\ \mathbf{U}\ \psi) \mid \mathbf{A}(\phi\ \mathbf{U}\ \psi) \end{array} \tag{4.1}$$

Intuitively, $\mathbf{EX}\ \phi$ means that ϕ holds on one of K's next states. $\mathbf{E}(\phi\ \mathbf{U}\ \psi)$ means that there is one execution path in K's computation tree (rooted at K's innitial state), such that ψ eventually holds, and until then ϕ holds (along the path).

$\mathbf{A}(\phi\ \mathbf{U}\ \psi)$ means as with **EU**, but the until-property has to hold on all possible paths that branch out from K's execution tree (rooted at its innitial state).

To define these formally, we will first introduce some auxilliary notations. We write $K \models \phi$ to mean that ϕ is valid CTL property of the Kripke structure K. We write $K, t \models \phi$ to mean that ϕ is a valid CTL property on the computation tree t, where t is an computation tree of K.

$paths(t)$ is the set of all paths through the tree t, which starts at its t's root.

If s is a state in K, $tree(s)$ is the a computation tree (of K), rooted in s.

We define:

$$K \models \phi \quad = \quad K, tree(s_0) \models \phi \tag{4.2}$$

where s_0 is K's innitial state.

The meaning of various CTL formulas are formally defined as follows:

1. If p is an atomic proposition (from K's *Prop*), it holds on a computation tree t if it holds on t's root:

$$K, t \models p \quad = \quad p \in V(root(t))$$

2. $\neg\phi$ holds on a computation tree t, if ϕ does not hold:

$$K, t \models \neg\phi \quad = \quad \text{not } (K, t \models \phi)$$

3. $\phi \wedge \psi$ holds on t if each holds individually:

$$K, t \models (\phi \wedge \psi) \quad = \quad (K, t \models \phi) \text{ and } (K, t \models \psi)$$

4. **EX** ϕ holds on t if ϕ holds on the tree rooted at one of t's root next-state:

$$K, t \models \mathbf{EX}\ \phi \quad = \quad \text{there is a state } v \in R(root(t)) \text{ such that } K, tree(v) \models \phi$$

   This is called the 'exists-next' operator.

5. $\mathbf{E}(\phi\ \mathbf{U}\ \psi)$ holds on t if there is one path π, on which the until-property holds:

$$\begin{aligned} K, t \models \mathbf{E}(\phi\ \mathbf{U}\ \psi) \quad = \quad & \text{there is a } p \in paths(t) \text{ and a } j \geq 0 \text{ such that:} \\ & K, tree(p_j) \models \psi \\ & \text{and, for all } i,\ 0 \leq i < j: \quad K, tree(p_i) \models \phi \end{aligned}$$

   This is called the 'exists-until' operator.

6. $\mathbf{A}(\phi\ \mathbf{U}\ \psi)$ holds on t if the until-property holds on all paths starting at t's root:

$$\begin{aligned} K, t \models \mathbf{A}(\phi\ \mathbf{U}\ \psi) \quad = \quad & \text{for all } p \in paths(t), \text{ there is a } j \geq 0 \text{ such that:} \\ & K, tree(p_j) \models \psi \\ & \text{and, for all } i,\ 0 \leq i < j: \quad K, tree(p_i) \models \phi \end{aligned}$$

   This is called the 'always-until' operator.

Notice that the 'E' and 'A' can be seen as a separate operator that quantifies over $paths(t)$. The first existentially quantifies over it, and the second universally.

The 'X' and 'U' have the same intuition as in LTL. However the above syntax forbids you to write nested 'until' within the same 'E' or 'A' quantifier. E.g. this has no meaning in CTL:

$$\mathbf{E}(p\ \mathbf{U}\ (q\ \mathbf{U}\ r))$$

Furthermore note that if a computation tree t has v as its root, then it is the *only* computation tree rooted at v. In otherwords, the root and the tree fully determines each other. So, let us add this notation:

$$K, v \models \phi \quad = \quad K, tree(v) \models \phi$$

In literature, CTL is often defined in terms of the root of the corresponding computation tree; so now you know what is meant.
We can furthermore define a number of handy derived operators:

- Disjunction and implication:

$$\begin{aligned}\phi \vee \psi &= \neg(\neg\phi \wedge \neg\psi)\\ \phi \rightarrow \psi &= \neg\phi \vee \psi\end{aligned}$$

- Always-next ϕ:

$$\mathbf{AX}\,\phi = \neg(\mathbf{EX}\,\neg\phi)$$

- Exists-eventually and always-eventually:

$$\mathbf{EF}\,\phi = \mathbf{E}(\texttt{true}\ \mathbf{U}\ \phi)$$

$$\mathbf{AF}\,\phi = \mathbf{A}(\texttt{true}\ \mathbf{U}\ \phi)$$

- Exists-globally and always-globally:

$$\mathbf{EG}\,\phi = \neg(\mathbf{AF}\,\neg\phi)$$

$$\mathbf{AG}\,\phi = \neg(\mathbf{EF}\,\neg\phi)$$

Actually we only need either the exist-until or the always-until (rather than both) as a primitive operator, since one can be expressed in terms of the other. We have this equality:

$$\mathbf{A}(\phi\ \mathbf{U}\ \psi) = \neg\mathbf{E}(\neg\psi\ \mathbf{U}\ (\neg\phi \wedge \neg\psi))\ \wedge\ \neg\mathbf{EG}\,\neg\psi \tag{4.3}$$

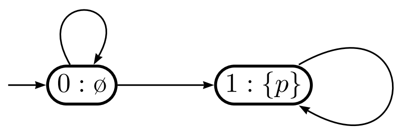
4.1.1 CTL vs LTL

Some properties can be expressed in both CTL and LTL, e.g:

CTL	LTL
$\mathbf{AG}\,p$	$\Box p$
$\mathbf{AF}\,p$	$\Diamond p$
$\mathbf{AX}\,p$	$\mathbf{X}\,p$
$\mathbf{A}(p\ \mathbf{U}\ q)$	$p\ \mathbf{U}\ q$

where p and q are atomic propositions.

Some CTL properties cannot be expressed in LTL. For example, consider the automaton below, with $Prop = \{p\}$.



The CTL property **EF** p holds on this automaton, which means that there is one execution on which the proposition p will eventually hold.

We cannot express this in LTL. E.g. The closest we have in LTL is $\Diamond p$, but it would require that p holds on *all* executions, which is too strong, and is not valid for the above automaton. Someone may suggest to try to express it with negation: $\neg\Box\neg p$; but this just the equivalent of $\Diamond p$. So that won't work either.

Essentially LTL cannot express the property because it cannot existentially quantify over the set of all executions. An LTL property is always defined with respect to *all* executions.

Some LTL properties cannot be expressed in CTL. Consider this automaton:

$$\begin{tikzcd}
\arrow[r] & 0 : \{p\} \arrow[loop above] \arrow[r] & 1 : \emptyset \arrow[r] & 2 : \{p\} \arrow[loop right]
\end{tikzcd}$$

This LTL property is valid on this automaton: $\Diamond\Box p$. It says, over any execution of this automaton, along the execution eventually p will continue to hold. There is a nested quantinfications over each execution here.

However CTL has no mechanism to express such a nested quantifications over a single path. The closest formula in CTL to express $\Diamond\Box p$ is $\mathbf{AF}\ (\mathbf{AG}\ p)$. But this requires that all execution paths from state-0 satisfies the $\mathbf{F}(\mathbf{AG}\ p)$ part. In particular, consider the infinite path where we just keep cyling in state-0. Let's call this path π. By the definition of **AF**, there should be some $i \geq 0$, such that $tree(\pi_i) \models \mathbf{AG}\ p$. But $\pi = 0, 0, 0, ...$ So, its i-th state is just state-0, and $\mathbf{AG}\ p$ does *not* hold on $tree(0)$!

4.1.2 CTL*

CTL* is a superset of both CTL and LTL. The syntax is below. A CTL formula is a state formula, which in turns is built from path fomulas.

The syntax of state formulas:

$$\begin{aligned} \phi \quad ::= \quad & p, \quad \text{where } p \text{ is an atomic proposition} \\ & \mid \neg\phi \mid \phi \wedge \psi \\ & \mid \mathbf{E}f \mid \mathbf{A}f \end{aligned} \tag{4.4}$$

where f is a *path* formula, with the following syntax:

$$\begin{aligned} f \quad ::= \quad & \phi, \quad \text{where } \phi \text{ is a state formula} \\ & \mid \neg f \mid f \wedge g \\ & \mid \mathbf{X}\ f \mid f\ \mathbf{U}\ g \end{aligned} \tag{4.5}$$

In CTL* we can express e.g. $\mathbf{E}(p\ \mathbf{U}\ (p\ \mathbf{U}\ q))$.

CTL* is decidable.

4.1.3 Expansion Laws

Let's first consider LTL's 'until' operator, because it is a bit easier to explain. It satisfies this 'expansion' property:

$$\phi\ \mathbf{U}\ \psi \quad = \quad \psi \vee (\phi \wedge \mathbf{X}\ (\phi\ \mathbf{U}\ \psi)) \tag{4.6}$$

It says, $\phi\ \mathbf{U}\ \psi$ holds if ψ holds immediately, or if ϕ holds now the $\phi\ \mathbf{U}\ \psi$ holds with respect to all executions from one-step ahead. Notice the recursive relation here.

CTL's 'until', and all its derived operators, also has a similar property, as shown below. We will see an application of the property in the next subsection.

Theorem 4.1.1 : CTL's Expansion Property

- $\mathbf{E}(\phi\ \mathbf{U}\ \psi) \quad = \quad \psi \vee (\phi \wedge \mathbf{EX}\ (\mathbf{E}(\phi\ \mathbf{U}\ \psi)))$
- $\mathbf{A}(\phi\ \mathbf{U}\ \psi) \quad = \quad \psi \vee (\phi \wedge \mathbf{AX}\ (\mathbf{A}(\phi\ \mathbf{U}\ \psi)))$
- $\mathbf{EF}\ \phi \quad = \quad \phi \vee \mathbf{EX}\ (\mathbf{EF}\ \phi)$
- $\mathbf{AF}\ \phi \quad = \quad \phi \vee \mathbf{AX}\ (\mathbf{AF}\ \phi)$
- $\mathbf{EG}\ \phi \quad = \quad \phi \vee \mathbf{EX}\ (\mathbf{EG}\ \phi)$

$\Box$

4.1.4 CTL Model Checking

Consider a Krikpe structure $K = (S, s_0, R, Prop, V)$. Let's treat a CTL formula as a predicate on K's states. Let's write W_ϕ to mean the set of K's states such that ϕ holds. That is:

$$W_\phi \;=\; \{s \mid s \in S \text{ and } K, s \models \phi\}$$

Notice that we then have this relation:

$$K \models \phi \;=\; s_0 \in W_\phi \tag{4.7}$$

Well, this gives an idea as to how to check whether ϕ is a valid property of K. The above says that you can thus proceed by first calculating the set W_ϕ, and then we simply check whether $s_0 \in W_\phi$.

So now the problem is reduced to calculating W. For the formulas that contains no 'until' operator it is straight forward:

1. If p is an atomic proposition, calculating W_p is straight forwards:

   $$W_p \;=\; \{s \mid s \in S \text{ and } p \in V(s)\}$$

2. If W_ϕ has been calculated, calculating $W_{\mathbf{EX}\ p}$ is straight forward:

   $$W_{\mathbf{EX}\ p} \;=\; \{s \mid s \in S \text{ and } R(s) \cap W_\phi \neq \emptyset\}$$

   which simply says that we take all states s (of K) that has at least one successor in W_ϕ.

3. If W_ϕ has been calculated, calculating $W_{\mathbf{AX}\ p}$ is also straight forward:

   $$W_{\mathbf{AX}\ p} \;=\; \{s \mid s \in S \text{ and } R(s) \subseteq W_\phi\}$$

   which says that we take all states s (of K) that has *all* its successors in W_ϕ.

Calculating W of always-until

To calculate the W of 'until', we are helped by its expantion property —see Theorem 4.1.1. It implies that the same relation holds for W of 'until'. So (let's take the 'always-until' variant):

$$W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)} \;=\; W_\psi \cup (W_\phi \cap \{s \mid s \in S \text{ and } R(s) \subseteq W_{\mathbf{A}(p\ \mathbf{U}\ q)}\})$$

This is a bit long formula. Let's abbreviate: $Z = W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)}$. Then the equation becomes:

$$Z \;=\; W_\psi \;\cup\; \underbrace{(W_\phi \cap \{s \mid s \in S \text{ and } R(s) \subseteq Z\})}_{f(Z)}$$

If we assume that we have already calculated W_ϕ and W_ψ, then all elements at the right hand side of the equation above, except for Z, are known. So, very abstractly the above equation has this form:

$$Z \;=\; W_\psi \;\cup\; f(Z)$$

where f is a function abbreviates the part of the original right hand side as indicated above.

We are basically looking for the *smallest solution* of the above equation. Note that e.g. the entire S is trivially a solution, but that is not the one we mean.

To calculate the smallest solution for Z we will proceed iteratively as follows. We will approximate Z with a series $Z_0, Z_1, Z_2, \ldots$. They are calculated as follows:

1. Start with an empty approximation:

   $$Z_0 \ = \ W_\psi$$

2. The next approximation is calculated from the previous one, as follows:

   $$Z_{i+1} \ = \ f(Z_i) \ \cup \ Z_i$$

3. Keep approximating, until we find a $Z_{k+1} = Z_k$. Then Z_k is equal to the Z we are looking for.

Proof:

Observe that every iteration grows the size of Z_i:

$$Z_{i+1} \ \supseteq \ Z_i$$

Because K is a finite state automaton (S is finite), we cannot indefinitely grow the approximation. So, the algorihtm must terminate.

Due to the way f is defined, oberserve that Z_i consists of those states from which we can reach a state satisfying ψ in at most i-steps, while still satisfying the 'until' constraint.

Because the algotirhm terminates (we just argued it above), then it must terminate with Z_i includes all the states in the solution Z.

□

Calculating W of exists-until

Analogously with 'always-until' above, based on the expansion property of 'exists-until', we have:

$$W_{\mathbf{E}(\phi \ \mathbf{U} \ \psi)} \ = \ W_\psi \cup (W_\phi \cap \{s \mid s \in S \text{ and } R(s) \cap W_{\mathbf{A}(p \ \mathbf{U} \ q)} \neq \emptyset\})$$

Let's abbreviate: $Z = W_{\mathbf{E}(\phi \ \mathbf{U} \ \psi)}$. Then the equation becomes:

$$Z \ = \ W_\psi \ \cup \ \underbrace{(W_\phi \cap \{s \mid s \in S \text{ and } R(s) \cap Z \neq \emptyset\})}_{g(Z)}$$

Or:

$$Z \ = \ W_\psi \ \cup \ g(Z)$$

We are looking for the smallest solution for this Z. We will calculate it iteratively as before. We will approximate Z with a series $Z_0, Z_1, Z_2, \ldots$. They are calculated as below. It is the same algorithm as in always-until. The only difference is that we need to use g instead of f:

1. Start with an empty approximation:

   $$Z_0 \ = \ W_\psi$$

2. The next approximation is calculated from the previous one, as follows:

   $$Z_{i+1} \ = \ g(Z_i) \ \cup \ Z_i$$

3. Keep approximating, until we find a $Z_{k+1} = Z_k$. Then Z_k is equal to the Z we are looking for.

The above described CTL model checking algorithm has the time complexity of $O((N+E) * |\phi|)$ where N is the number of states in the target Kripke structure, E is its number of arrows, and $|\phi|$ is the size of the CTL formula we are verifying.

4.2 References

- C. Baier and J. P. Katoen. *Principles of Model Checking.* MIT Press, New York, May 2008

Chapter 5

Symbolic Model Checking and BDD

5.1 Symbolic Representation of State Space

As you can see in the previous model checking algorithms, both for LTL and CTL, we need to have the 'state space' of our target program (or model) in order to apply the algorithm. LTL model checking allows you to lazily construct this state space, but ultimately you need to construct it completely.

Now, the symbolic model checking approach proposes to represent the state space symbolically with a formula. It would be a big formula, but nevertheless you can store in less space than the state space itself. Essentially, the idea is to represent multiple states and multiple arrows in the state space with just a single small formula; this saves space.

However, this does mean that our model checking algorithms have to be adapted so that they work on formulas, rather than an automaton. We will later see how this works, but first let us see how we can symbolically represent an automaton (which is what a state space is) with formulas.

To be more precise, we will represent automatons with *Boolean functions*. Such a function $f(x, y)$ takes parameters of Boolean type, and returns a Boolean value. The body of this function is described by a Boolean formula, with the following syntax:

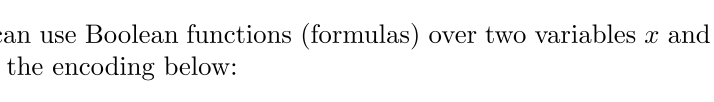
$$p \quad ::= \quad 0 \ \mid \ 1 \ \mid \ \mathit{variable} \mid \neg p \mid p \wedge q \mid p \vee q$$

For brevity we will use 1/0 instead of `true` and `false`. We also just write $x.y$ or even xy (if it is clear that x and t are two variables) to mean $x \wedge y$. We write $\bar{x}\bar{y}$ and $\overline{xy}$ to mean $\neg x \wedge \neg y$ and respectively $\neg(x \wedge y)$.

We will use derived Boolean operators = (equality on Boolean values) and $\Rightarrow$, and furthermore this quantifications over Boolean values:

$$\begin{array}{lcl} (\exists x :: p) & = & p[1/x] \vee p[0/x] \\ (\forall x :: p) & = & \neg(\exists x :: \neg p) \end{array}$$

Consider now this automaton K with just four states:


$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[loop above] \arrow[r, bend right] & 1 \arrow[loop above] \arrow[l, bend right] \arrow[r] & 2 \arrow[loop above] \arrow[r, bend right] & 3 \arrow[loop right] \arrow[l, bend right]
\end{tikzcd}$$

We can use Boolean functions (formulas) over two variables x and y to encode its four states; e.g. with the encoding below:

encoding	state
$\overline{xy}$	0
$\bar{x}y$	1
$x\bar{y}$	2
xy	3

The nice thing about this is that we can also use formulas to represent *sets* of states. E.g. the formula x represents the set $\{2,3\}$; the formula $\bar{x}$ represents $\{0,1\}$; the formula $x \vee y$ represents $\{1,2,3\}$.

An arrow relates a state to a possible next-state. If we use a primed names of x and y to represent next-state, we can also encode arrows. E.g:

encoding	arrow
$\overline{xy}\,\overline{x'y'}$	the loop from 0 to 0
$\overline{xy}\bar{x}'y'$	the arrow from 0 to 1

We can also encode multiple arrows with a single Boolean function, e.g. the two arrows going into state 0 can be encoded by:

$$\bar{x}\overline{x'y'}$$

It can be quite short, thus saving space. E.g. all the four arrows between state 2 and 3 can be represented by:

$$xx'$$

The entire automaton can be described by the following Boolean function:

$$R(x,y,x',y') \quad = \quad \bar{x}\overline{x'} \ \vee \ \bar{x}y\overline{y'} \ \vee \ xx'$$

We can easily extend this to encode the propositions that hold in the states. For example, suppose label the states of the automaton that we had before as follows; *Prop* $= \{p,q\}$ is assumed:

$$\begin{tikzcd}
{} \arrow[r] & 0:\emptyset \arrow[loop above] \arrow[r, bend right] & 1:\emptyset \arrow[loop above] \arrow[l, bend right] \arrow[r] & 2:\{p\} \arrow[loop above] \arrow[r, bend right] & 3:\{p,q\} \arrow[loop right] \arrow[l, bend right]
\end{tikzcd}$$

We see that p holds on states 2,3, whereas q only holds on 3. We can capture this with the following Boolean function:

$$V(x,y,p,q) \quad = \quad (p = x) \wedge (q = xy)$$

where $f = g$ above is just boolean equality; so it holds if either fg or $\bar{f}\bar{g}$.

Question: so how do we handle automaton with concrete states? So, we don't have labeling with propositions. Instead, we have variables that take values. E.g. variables a, b which in every state may take different values.

5.2 CTL Symbolic Model Checking

Let $K = (S, s_0, R, Prop, V)$ be a Krikpe structure. Let ϕ be a CTL formula we want to verify on K. Recall our model checking approach proceeds by first calculating W_ϕ (the set of all states of K that would satisfy ϕ), and then verify $s_0 \in W_\phi$.

We need to turn this to work on Boolean functions. You have seen that a set of states can be expressed by a Boolean function. Suppose now, as in the above example, that we can use two Boolean variables x and y to encode the states of K. We will first construct a symbolic representation of W_ϕ. This is a Boolean function over x and y, which we will just denote by:

$$W_\phi(x,y)$$

Intuitively this function is as such that $s \in W_p hi$ if and only if $W_p hi(enc(s)) = 1$, where $enc(s)$ is the vector x, y that represents the state s.

Checking $s_0 \in W_p hi$ is now equivalent to checking if $W_\phi(enc(s_0)) = 1$. This can be easily generalized if we have multiple initial states, represented by a Boolean formula $init(x, y)$. Then we check if this formula is *valid*:

$$init(x, y) \;\wedge\; W_\phi(x, y)$$

Importantly, observe that this suggests that we can convert the problem of CTL model checking to the problem of evaluating a Boolean function, or the problem of checking validity in the proposition logic. The latter two problems are solvable.

This also suggests that if you can give a symbolical description of your target program, then we can model check it without having to explicitly construct its state space.

For the above to work, we still need to figure out a way to construct the symbolical representation of W_ϕ. We will show the construction below, through examples. Consider again the automaton K from the previous section, with $Prop = \{p, q\}$. In the section we have already given the symbolic description of this K, in the form of the functions:

$R(x, y, x', y')$	:	describes K's arrows
$V(x, y, p, q)$	:	describes the labelling with propositions

Atomic Proposition

Consider the atomic proposition p. W_p is then just the set of all K's states where p holds:

$$W_p \quad = \quad \{s \mid p \in V(s)\}$$

But we can also express this with a Boolean function:

$$W_p(x, y) \quad = \quad (\exists q :: V(x, y, 1, q))$$

Note however that strictly speaking $W_p(x, y)$ is a function, and not a set of states (whereas W_p is). It is a function that encodes the same set of states.

Suppose we want to verify if $K \models p$. This means checking whether $s_0 \in W_p$, which we can symbolically check by checking if this:

$$W_p(0, 0)$$

would evaluate to 1.

Note that $x = 0$ and $y = 0$ is the encoding of state 0, which is K's initial state in this example.

Conjunction and Negation

Well, assuming you have constructed the Boolean functions $W_\phi(x, y)$ and $W_\psi(x, y)$ then:

$$W_{\neg\phi}(x, y) \quad = \quad \neg W_\phi(x, y)$$

$$W_{\neg\phi\wedge\psi}(x, y) \quad = \quad W_\phi(x, y) \;\wedge\; W_\psi(x, y)$$

Next Property

Assume that we have already calculated $W_\phi(x,y)$. Now, $W_{\mathbf{EX}\ \phi}$ is:

$$W_{\mathbf{EX}\ \phi} \ = \ \{s \mid R(s) \ \cap \ W_\phi \neq \emptyset\}$$

We can express this with a Boolean function, namely:

$$W_{\mathbf{EX}\ \phi}(x,y) \ = \ (\exists x', y' :: R(x,y,x',y') \ \wedge \ W_\phi(x',y'))$$

The W of **AX** can be calculated indirectly through this equality: $\mathbf{AX}\ \phi = \neg\mathbf{EX}\ \neg\phi$. Or, if we want to express this directly:

$$W_{\mathbf{AX}\ p}(x,y) \ = \ (\forall x', y' :: R(x,y,x',y') \ \Rightarrow \ W_\phi(x',y'))$$

To verify e.g. $K \models \mathbf{EX}\ \phi$ we check $s_0 \in W_{\mathbf{X}\ p}$ instead. This is the same as checking this:

$$W_{\mathbf{EX}\ \phi}(0,0) \ = \ 1$$

Until

Well, assuming you have constructed the Boolean functions $W_\phi(x,y)$ and $W_\psi(x,y)$. To construct $W_{\mathbf{E}(p\ \mathbf{U}\ q)}$ we iteratively construct a sequence of approximations Z_i, where:

$$Z_0 \ = \ W_\psi$$

$$Z_{i+1} \ = \ ((W_\phi \cap \{s \mid R(s) \ \cap \ Z_i \neq \emptyset\}) \ \cup \ Z_i$$

But we can also represents Z_i with Boolean functions:

$$Z_0(x,y) \ = \ W_\psi(x,y)$$

$$Z_{i+1}(x,y) \ = \ (W_\phi(x,y) \wedge (\exists x', y' :: R(x,y,x',y') \wedge Z_i(x',y'))) \ \vee \ Z_i(x,y)$$

However, we have to stop the iteration if we have reach a fix point. So, a point where $Z_{i+1} = Z_i$. If we have set of states then we know how to do it. However, now we have Boolean functions. To check if you have reached the fix point you would then need to check whether:

$Z_i(x,y)$ and $Z_{i+1}(x,y)$ are equivalent

And you need to be able to do this efficiently.

The all-until operator can be handled analogously.

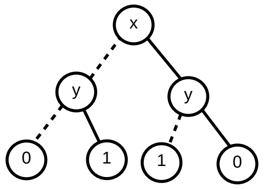
5.3 Representing Boolean Function

One way to check if two Boolean formulas p and q are equivalent is to turn this to a satisfiability problem. So, check whether $p \neq q$ is satisfiable. If so, then they are not equivalent; else they are. You can use a SAT solver to do this. Note that this problem is in general NP-complete.

We will look at a different approach. We will construct some form of 'standard' representations of proposition formulas, in such a way that p and q are equivalent if and only if their standard representations are structurally the same. So, we can instead compare their representations. Such a representation is also called *canonical representation* or *normal form*.

Truth table is such a canonical representation, but it won't do for use, due to the exponential size of the table.

A *binary decision tree* is a tree whose nodes are labelled with a variable name, and leaves labelled with either 0 or 1. Each node has two child-nodes: its *low* and respectively *high* child. Here the low child is indicated by the dashed line connecting to it. See the example below:



For later, it is easier if we just treat a leaf as a special kind of node. So, a tree has nodes, but some of these nodes are leaves.

If v is not a leaf, we will assume the functions $low(v)$ and $high(v)$ that return v's low and respectively high child. The function $ind(v)$ returns the 'variable name' that decorates the node. If v is a leaf, $val(v)$ returns its Boolean value, which is either 0 or 1.

We will require that along every path in the tree, each variable name occurs at most once.

Such a tree can be used to represent a Boolean function. Given a tree, the function represented by it can be recursively reconstructed as follows:

1. If v is a leaf, $func(v) = val(v)$.
2. If v is a non-leaf node, decorated with x (so, $ind(v) = x$) then:

$$func(v) \quad = \quad \bar{x}.func(low(u)) \;\vee\; x.func(high(u))$$

3. If t is a tree with root v_0, the the formula t represents is $func(v_0)$.

For example, the Boolean function represented by the tree above is:

$$f(x,y) \;=\; func(root) \;=\; \bar{x}\bar{y}0 \;\vee\; \bar{x}y1 \;\vee\; x\bar{y}1 \;\vee\; xy0$$

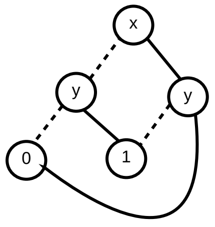
which can be simplified to:

$$f(x,y) \;=\; \bar{x}y \;\vee\; x\bar{y}$$

Every boolean function can be represented with a binary decision tree. Such a tree uses however a lot of space, since it requires exponential number of nodes, with respect to the number of parameters of our Boolean function, but we will improve this below.

A *binary decision graph* (BDD) is just like a *binary decision tree*, except that it is a *directed graph.* So, multiple nodes can share the same child or leaf. There are some additional constraints: the graph must have a single *root*, and all paths in the graph must end in the leaves.

Below is an example. To make the drawing less verbose, I keep the direction of the edges in my drawings implicit (it's a *directed* graph): it is always from top to bottom. The root is always the topmost node.



A BDD can be used to represent a Boolean function as well. We use the same algorithm as for the tree to reconstruct the function that the graph represents. If G is a BDD with root v_0, then $f(G)$ is just $f(v_0)$.

Because in a graph you can share nodes, it is more space-efficient than a tree. The above graph represents the same Boolean function as the tree you earlier saw:

$$f(x,y) \;=\; func(root) \;=\; \bar{x}y \;\vee\; x\bar{y}$$

but it uses two less nodes.

A BDD is called *reduced* if it cannot be made smaller (without changing its meaning as a Boolean function); see below for the formal definition. For example, the above BDD is reduced. In terms of space usage, it would be nice to have a reduced BDD.

Definition 5.3.1 : Reduced BDD
A BDD G is reduced if:

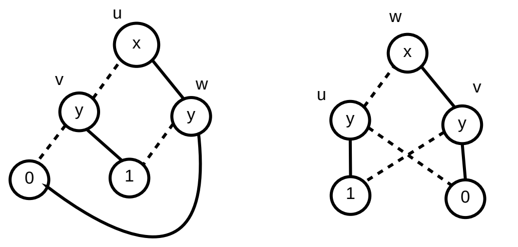
- for any non-leaf node v, $low(v) \neq high(v)$. Otherwise G can be simplified.
- for any distinct nodes v and v', the subgraphs rooted at them are *not* isomorphic. Otherwise G can be simplified. The meaning of 'isomorphic' is explained below.

□

Two graphs may look different, but yet structurally the same. They are then called isomorphic. More precisely:

Definition 5.3.2 : Isomorphic BDD
Two BDDs G and H are isomorphic if we can obtain H from G by renaming G's nodes, and vice versa. You are not allowed to rename $var(v)$ nor $val(v)$.
□

For example, the two BDDs below are isomorphic:



Note that since a BDD may only have one root, and the children of any node are distinguished (low and high), the choice for σ, that map nodes of G to those of H, is actually quite constrained. It must map G's root to H's root, and the root's low (high) child in G to the corresponding root's low (high) child in H, and so on all the way down to the terminal nodes. Hence, testing isomorphism on BDDs is quite simple; it can be done in $O(N)$, where N is the number of nodes in G or H, whichever the smallest one is.

Suppose we introduce a certain linear ordering among the 'variables' (the decorations of the nodes in a BDD), e.g. x, y, z. You could also choose y, x, z; what matters now is that you fix an ordering. If x and y are two variables, we will write $x \prec y$ to mean that in this ordering x preceeds y.

Definition 5.3.3 : Ordered BDD
An *ordered* BDD (OBDD) is a BDD G such that the variables along any path in the BDD respect the ordering. More precisely, for any non-leaf node v:

$$ind(v) \prec ind(low(v)) \quad \text{and} \quad ind(v) \prec ind(high(v))$$

□

For example, if x, y is our ordering, then all the above BDDs are ordered, and else not.

An important propertry of OBDD is the following:

Theorem 5.3.4 :
If you fix the ordering of the variables, then for every Boolean function f there is a unique (modulo isomorphism) reduced OBDD (with respect to the given ordering) that represents this f.
□

The 'modulo isomorphism' here means that if there are multiple reduced OBDDS representing f, they will all be isomorphic to each other.

To put it differently, OBDD can be used as the cannonical representation for Boolean formulas. Note however, the representation is bound to the variable ordering you chosed. If you change the ordering, you may get a different OBDD representation. Some orderings yield the optimal (smallest) OBDD, some may do the opposite.

The above theorem implies that to check whether two formulas f and g are equivalent, we can do it by comparing if their OBDDs are isomorphic

5.3.1 Checking Satisfaction

A Boolean function f is *satisfiable* if it is possible to find values for its parameters that would make the function to return 1. The function is valid if for all possible values of its parameters, it always returns 1. If we can check satisfiability, we can also check validity: f is valid if and only if $\neg f$ is not satisfiable.

To check if f is satisfiable, we can first construct its OBDD F, and then checks whether the leaf 1 is reachable from F's root. You have seen before that you can use DFS to check reachability. Moreover, DFS will give you the path that would leads to from the root to 1; this path tells you how to assign values to the parameters of f to give you the 1.

5.3.2 Constructing OBDD

Given a formula p, we still need an algorithm that would construct its OBDD. The naive way to do it is to first construct a binary decision tree for it, and then reduce it. But this won't do, since the tree is exponential in size.

We can instead construct this incrementally, by combining the BDDs of the subformulas of p.

Reduce

This algorithm is used to reduce an OBDD. The input of the algorithm is an OBDD G. It returns a OBDD H, such that it represents the same Boolean function as G, and furthermore H is a reduced OBDD.

The idea is to traverse G from bottom to top. At each node v, we check if we have seen another node v' that represent the same formula as v. Those nodes are thus equivalent, and furthermore v is thus redundant, and won't be copied to H. Else it will.

To keep track of which nodes are equivalent, we will maintain a mapping ID. It maps the nodes of G that we have visited, to the nodes of H. If two nodes u and v map to the same w, then they are all equivalent, but we have chosen to pick w to be put in the reduced graph H[1].

We will use the notation $ID(v) = w$ to mean that ID maps v to w. We write $v \in ID$ to mean that ID contains a mapping for v (it maps v to something).

For explaining the algorithm, it is easier if we capture assumptions on ID a bit more formally with invariants, which the algorithm will maintain later.

The fact that ID maps to H is expressed by this invariant:

$$I_0 : \quad v \in ID \ \text{ implies } \ ID(v) \in H \tag{5.1}$$

[1]To explain the name "ID", it is because that is how it is called in the original algorithm [3]. However, the original ID maps a node to a unique number, which can be though as a new Identity for the node. We use ID differently, because this is easier. However, abstractly we are still doing the same thing. We still keep the name ID to make it easier for you to relate this account of the algorithm to the original one.

If $ID(u) = v$ then these nodes are equivalent; thus they represent the same Boolean functions. This is expressed by this invariant:

$$I_1: \quad u \in ID \text{ implies } func(u) = func(ID(u)) \tag{5.2}$$

For sanity, we add these invariants too:

$$I_2: \quad u \in H \text{ implies } ID(u) = u \tag{5.3}$$

$$I_3: \quad v \in H \text{ and } v \text{ is non-leaf implies } low(v), high(v) \in H \tag{5.4}$$

The algorithm proceeds as follows:

1. We innitialize the mapping ID to empty.
2. For every leaf v in G we do:
   (a) If H contains no leaf of the same value as v, we add v to H. We add an identity-entry $v \mapsto v$ to the mapping ID.
   (b) If H already contains a leaf v' of the same value ($val(v) = val(v')$), then v is redundant. We do *not* add v to H. We do add the entry $v \mapsto v'$ to the mapping ID.
3. We proceed recursively, along the index-numbers of the variables decorating G's nodes. We process the high index first, then move to the lower ones. Visually, this is as if we start from the bottom of the graph and proceed towards the top.

   Now, assume all nodes $u \in G$ such that $ind(u) > i$ have been processed.

   We proceed processing the nodes v with $ind(v) = i$. Note first that the Boolean formula represented by v is:

$$func(v) \;=\; ind(v).func(high(v)) \;\vee\; \overline{ind(v)}.func(low(v))$$

   For every such v, note that its children $low(v)$ and $high(v)$ would have higher indices than v. Therefore they are in ID. Next, we do:

   (a) If $ID(high(v)) = ID(low(v))$, it follows from I_1 that:

$$func(high(v)) \;=\; func(low(v)) \;=\; func(ID(low(v)))$$

   And therefore:

$$\begin{aligned} func(v) &= ind(v).func(high(v)) \;\vee\; \overline{ind(v)}.func(low(v)) \\ &= ind(v).func(low(v)) \;\vee\; \overline{ind(v)}.func(low(v)) \\ &= func(low(v)) \\ &= func(ID(low(v))) \end{aligned}$$

   H must have already contained $ID(low(v))$, due to I_0. But the Boolean formula represented by v is just the same as that represented by $ID(low(v))$. So, v is redundant! Therefore we do *not* add it to H.

   We do add the entry $v \mapsto ID(low(v))$ to ID.

   (b) If H contains v' decorated with the same variable-name as that of v (so, $ind(v') = ind(v)$), and:

$$ID(low(v)) = low(v') \quad \text{and} \quad ID(high(v)) = high(v')$$

   Then it follows, by I_2, that:

$$f(low(v)) = f(low(v')) \quad \text{and} \quad f(high(v)) = f(high(v'))$$

Therefore:

$$\begin{aligned} f(v) &= ind(v).f(high(v)) \ \vee \ \overline{ind(v)}.f(low(v)) \\ &= ind(v').f(high(v)) \ \vee \ \overline{ind(v')}.f(low(v)) \\ &= ind(v').f(high(v')) \ \vee \ \overline{ind(v')}.f(low(v')) \\ &= f(v') \end{aligned}$$

So, the Boolean formula represented by v is just the same as that represented by v'. Then v is redundant. We do not add it to H.

We do add the entry $v \mapsto ID(v')$ to ID.

(c) If neither (a) nor (b) holds, then v is not redundant. It is added to H, and the entry $v \mapsto v$ is added to H.

Furthermore, we change the children of v:

- set $v.low$ to $IH(v.low)$, and
- set $v.high$ to $IH(v.high)$, and

For a more efficient work out of this algorithm, see [3], which is $O(|G| * log|G|)$.

Apply

If we already have the OBDDs for f and g, we can combine them to construct an OBDD for $f \wedge g$. We will give an general algorithm to do this for all Boolean binary operators. It even works for $\neg$ because $\neg p = p$ **xor** 1. Let $\otimes$ below be any Boolean binary operator.

Let f and g be two Boolean functions over the same set of parameters, represented by OBDD F and respectively G. Let x be one of these parameters. First, notice that we have this equality:

$$f \otimes g \quad = \quad x.(f[1/x] \otimes g[1/x]) \ \vee \ \bar{x}.(f[0/x] \otimes g[0/x]) \tag{5.5}$$

where $f[1/x]$ means the function we would obtain if we replace x with 1.

The above suggests that we can perhaps try to construct the OBDD of $f \oplus g$ recursively.

The algorithm will be called $apply(F, G, \otimes)$. It takes the OBDDs F and G as inputs, and the operation to apply. It will return a new OBDD, such that (specification of 'apply'):

$$func(F) \oplus func(G) \quad = \quad func(apply(F, G, \otimes))$$

Suppose r_F and r_G are the roots of F and respectively G. We will write $low(F)$ to denote the subgraph of F, rooted at $left(r_F)$. Similarly, we define $high(F)$.

We have a number of cases:

1. If F and G both turn out to only contain one leaf. So, they represent the constant functions, which are either 0 or 1.

   In this case, we directly calculate $val(r_F) \otimes val(r_G)$, and construct the BDD representing the result.

2. Else, at least one of the two graphs are non-trivial. We have these cases:

   (a) The roots r_F and r_G are decorated with the same variable (so, $ind(r_F) = ind(r_G)$); suppose that this variable is x. Now, according to the equality above we have:

   $$f \otimes g \quad = \quad x.(f[1/x] \otimes g[1/x]) \ \vee \ \bar{x}.(f[0/x] \otimes g[0/x])$$

   where $f = func(F)$ and $g = func(G)$.

   However, because x is the variable at the root, notice that:

   $$f[1/x] \ = \ func(high(F)) \quad \text{and} \quad f[0/x] \ = \ func(low(F))$$

And similarly for g. Therefore:

$$\begin{aligned}
func(F) \otimes func(G) &= x.(f[1/x] \otimes g[1/x]) \ \vee \ \bar{x}.(f[0/x] \otimes g[0/x]) \\
&= \begin{array}{l} x.(func(high(F)) \otimes func(high(G))) \\ \vee \\ \bar{x}.(func(low(F)) \otimes func(low(G))) \end{array} \\
&= x.apply(high(F), high(G), \oplus) \ \vee \ \bar{x}.apply(low(F), low(G), \oplus)
\end{aligned}$$

which shows how to recursively calculate the result of *apply*.

(b) Both r_F and r_G are decorated with a different variable, say x and y respectively, and suppose $x \prec y$ (the case when $y \prec x$ is symmetrical). Because G is an OBDD (*ordered* BDD), this implies that it has no node labelled with x. So, its function does not depend on x. So:

$$g[0/x] \ = \ g[1/x] \ = \ g$$

Therefore:

$$\begin{aligned}
func(F) \otimes func(G) &= x.(f[1/x] \otimes g[1/x]) \ \vee \ \bar{x}.(f[0/x] \otimes g[0/x]) \\
&= x.(func(high(F)) \otimes g) \ \vee \ \bar{x}.(func(low(F)) \otimes g) \\
&= x.apply(high(F), G, \oplus) \ \vee \ \bar{x}.apply(low(F), G, \oplus)
\end{aligned}$$

which shows how to recursively calculate the result of *apply*.

(c) One of r_F and r_G is a leaf, and the other is not. Suppose r_G is the leaf. This implies that $func(G)$ is a constant function, thus does not depend on x. So:

$$g[0/x] \ = \ g[1/x] \ = \ g$$

This is the same situation as we had in (b) above, and can be handled in the same way.

Now this algorithm, if implemented as is, it is very inefficient. E.g. consider subgraphs F' and G' of F respectively G. Eventually, we must recurse on them, thus calculating $apply(F', G', \oplus)$. However because we are dealing with graphs, F' and G' may have multiple parents, which will cause $apply(F', G', \oplus)$ to be called multiple times. Because this may happen again and again, as we recurse down the graphs, the overall complexity is exponential.

A simple trick to get around this to store and keep track that we have calculated $apply(F', G', \oplus)$. So the next time we need to do it again, we don't have to calculate, but can just retrieve the result from memory. This reduces the complexity to $O(|F| * |G|)$. For further tweaks, see [3].

Substitution

Let f be a Boolean function over a set of parameters. Let x be one of the parameters. Suppose we have already constructed an OBDD F that represents f. We now want to construct an OBDD representing $f[1/x]$, and analogouslt $f[0/x]$. Note that here x is not necessarily the root variable of F.

I'll leave this to you :)

Overview of complexity

Here is an overview of the complexity of various operations on OBDD. This is based on the original algorithms in [3]. Below, F and G are OBDDs; and let f and g be the Boolean functions represented them. $|F|$ is the number of nodes in F.

$satisfy(F)$	$O(\|F\|)$	// check if f is satisfiable
$reduce(F)$	$O(\|F\| * log\|F\|)$	
$apply(F, G, \otimes)$	$O(\|F\| * \|G\|)$	
$subst(F, x, constant)$	$O(\|F\| * log\|F\|)$	// construct and reduce the OBDD of $f[0/x]$ or $f[1/x]$

5.4 References

- Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986
- K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993

Bibliography

[1] P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, 2008.

[2] C. Baier and J. P. Katoen. *Principles of Model Checking.* MIT Press, New York, May 2008.

[3] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[4] E. Lehmann and J. Wegener. Test case design by means of the CTE XL. In *Proceedings of the 8th European Int. Conference on Software Testing, Analysis & Review (EuroSTAR)*, 2000.

[5] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[6] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[7] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

Index