Lecture Notes Program Verification, v2013.0

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

email: S.W.B.Prasetya@uu.nl
URL: www.cs.uu.nl\~wishnu.html

Contents

1 Testing **5**
- 1.1 Introduction . . . 5
- 1.2 Test Adequacy . . . 7
  - 1.2.1 Path-based coverage . . . 9
  - 1.2.2 Linearly Independent Paths . . . 13
- 1.3 White box and black box testing . . . 14
  - 1.3.1 Classification tree for black box testing . . . 14
- 1.4 Regression . . . 16
  - 1.4.1 Test cases selection . . . 16
- 1.5 Literature . . . 18

2 Hoare Logic **19**
- 2.1 Examples of Hoare logic's inference rules . . . 19
- 2.2 Weakest pre-condition calculation . . . 20
- 2.3 Inference rule for loop . . . 20
- 2.4 Invariant as abstraction . . . 21
- 2.5 Rule for proving loop termination . . . 21
- 2.6 Abstract model of programs . . . 22
- 2.7 uPL . . . 23
- 2.8 Some Commonly Used Inference Rules and Theorems of Predicate Logic . . . 25
- 2.9 Inference Rules of Hoare Logic . . . 28
- 2.10 Esc/Java Core Logic . . . 29
  - 2.10.1 GCL . . . 30
  - 2.10.2 Representing objects in GCL . . . 30
  - 2.10.3 ESC/Java's GCL logic . . . 30
  - 2.10.4 Handling Loops . . . 31
- 2.11 Literature . . . 31

3 Model Checking **33**
- 3.1 Automata . . . 33
- 3.2 Some Basic Operations on Automata . . . 35
  - 3.2.1 Intersection of automata . . . 35
  - 3.2.2 Interleaving of automata . . . 36
- 3.3 Temporal Properties . . . 37
  - 3.3.1 Valid Property . . . 38
- 3.4 Buchi Automaton . . . 39
  - 3.4.1 Generalized Buchi Automata . . . 41
  - 3.4.2 Non-deterministic vs Deterministic Buchi Automata . . . 42
  - 3.4.3 Constructing Buchi Automaton . . . 42
- 3.5 LTL Model Checking . . . 42
  - 3.5.1 Emptiness of Buchi Automaton . . . 43
  - 3.5.2 By-passing the Intersection . . . 46

3.5.3 Model Checking on Concrete States . . . 46
3.6 Literature . . . 47

4 CTL Model Checking **49**
4.1 CTL . . . 49
4.1.1 CTL vs LTL . . . 51
4.1.2 CTL* . . . 52
4.1.3 Expansion Laws . . . 52
4.1.4 CTL Model Checking . . . 53
4.2 Literature . . . 55

5 Symbolic Model Checking and BDD **57**
5.1 Symbolic Representation of State Space . . . 57
5.2 CTL Symbolic Model Checking . . . 59
5.3 Representing Boolean Function . . . 61
5.3.1 Checking Satisfaction . . . 63
5.3.2 Constructing OBDD . . . 63
5.4 Literature . . . 67

6 CSP **69**
6.1 CSP Syntax and Primitive Operators . . . 69
6.1.1 Alphabet, External and Internal Events . . . 70
6.1.2 Parallel Composition . . . 71
6.2 Refinement and Specification . . . 71
6.2.1 Trace Semantic . . . 72
6.3 FSA Semantic . . . 73
6.3.1 Refinement Checking with FSA . . . 74
6.4 Enforcing Progress . . . 77
6.4.1 Refusals . . . 77
6.5 Refinement Checking Revisited . . . 79
6.5.1 Fixing the FSA-semantic . . . 79
6.5.2 Fixing the definition of initials . . . 80
6.5.3 Refusals of a state in FSA . . . 80
6.5.4 Fixing DFSA reduction . . . 81
6.5.5 Some notes . . . 83
6.6 Literature . . . 83

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 collected responses and states satisfy certain correctness expectations.

Ideally, we want the test-cases to be fully 'automated'[1]. That is, they can be executed by a machine without requiring human interactions. A large test-suite that requires many user interactions is expensive and prone to human mistakes. In some situations, this may unfortunately be the only way to test a system.

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. This can greatly influence your testing approach. For example testing an OO program requires a very different approach than testing a functional program. Testing an SUT with just a single operation is different than testing one with multiple operations. In the latter case, it also matters whether the operations can be called in any order, or whether they require specific orders.

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*.

[1] The term automated testing is unfortunately ambigous. It may be used to refer to automation in the test execution, or to refer to automatically generating test-cases.

'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.

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.

Maintenance

Wen you change your program, you may have to update your test-suite as well. While refactoring the signature of your methods may still be automatically reflected in your test-suite, changing the semantic of your program may entail a lot of work.

As said above, a test case checks if SUT's states and responses satisfy certain expectations. In testing jargon, these expectations are called *oracles*. Indeed, we can then 'simply' use P's specifications as these 'oracles'. However, this requires that the specifications are expressed in a machine readable form. In practice, people do not usually have complete informal specifications, let alone formal and machine readable ones.

Therefore, in practice oracles are often expressed in terms of concrete values, which are compared with SUT's responses or states, e.g. (in Pseudo code):

```
testcase1() {
  List s = [3,2,1] ;
  t = s.sort() ;              -- t is the SUT's reponse
  assert (t == [1,2,3])       -- [1,2,3] is our expected result --> concrete oracle
}
```

But note that concrete oracles have to be manually calculated for every test-case, and they are usually unique for each test-case. This is very very labour intensive, and creates obviously a serious maintenance problem: if the SUT's logic changes, we have to re-calculate the corresponding oracles.

Of course, you can slso write *partial* specifications to complement traditional test cases with concrete oracles. So that in case we are having problems in calculating the latter, we still have the partial specifications to fall back.

Testability

Your SUT is highly testable if its operations can be easily and systematically approached from a test case, e.g. if we can simply call them. For example, a pure Java class as the SUT is highly testable. This makes programming test cases much easier.

On the other hand, a GUI application is often much less testable. Graphic-based games (e.g. Doom, Mario, CoD) are even worse. Clicking on menus and buttons may require the mouse to be positioned on certain positions, and the SUT's reponses may have to be interpreted from the screen's bitmaps. This makes programming a test case much more difficult. There are so-called capture-and-replay tool that allows the tester to interact directly with the SUT. The interactions are recorded and can be replayed as a test-case. Thus, the tester does not have to calculate the click-coordinates herself. However, test-cases with concrete click-coordinates are still fragile. They won't work on devices with different screen sizes, and they will break when you change the layout of your GUI elements.

SUTs with low testability is painful to test. It can be mitigated by developing an additional layer that be turned on to programatically expose the operations. But such invesment may not come cheaply.

Non-determinism

When the SUT is deterministic, every test-case will trigger the same execution, and thus the same verdict, each time you execute it. Many programs are however non-deterministic, e.g. due to concurrency or interactions with non-deterministic external components (e.g. external services). When the degree of non-determinism is very high, *reproducibility* becomes a serious issue. When your test-case discovers a bug, to debug what causes it you need to be able to reproduce the execution that producuces the bug. With a non-deterministic SUT, this may take repeated runs of the test-case before you 'accidentally' manage to reproduce the execution. With a highly non-deterministic SUT, this can be very difficult.

This can be mitigated by bulding an 'overlay' that you can turn on to control or at least reduce the the non-determism. But again, this entails additional investment.

1.2 Test Adequacy

Testing is inherently incomplete, but it is practical and is thus 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. 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.

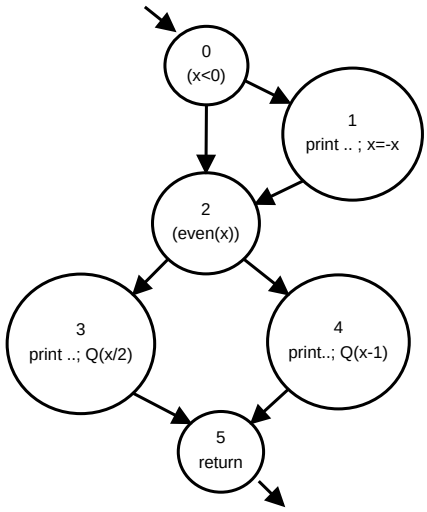
1.2.1 Path-based coverage

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[2]. 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.

[2]Note this is how *we* here define CFG. Depending on the purpose in the literature CFGs may be defined with higher or lower 'granularity'. E.g. for a different purpose you may want a node to represent a bytecode instruction.

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.

Let a CFG G be described by a tuple (N, E, S) 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. $S \in N$ is the starting node: the program represented by G always starts at this node. An *exit node* is a node with no outgoing arrow. G is assumed to have at least one exit node, which is reachable from S.

A 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 defined as:

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

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. For example, the program Q shown before has four maximal paths:

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

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)`

When a test case is executed, the execution will tarverse some path in the program's CFG. Let's call this *execution path*. If the execution terminates, its path is thus maximal. We define:

Definition 1.2.1 : Tour
A path τ tours another path σ if σ is a subpath of τ. That is, if:

$$\tau \;=\; t +\!\!+ \sigma' +\!\!+ u$$

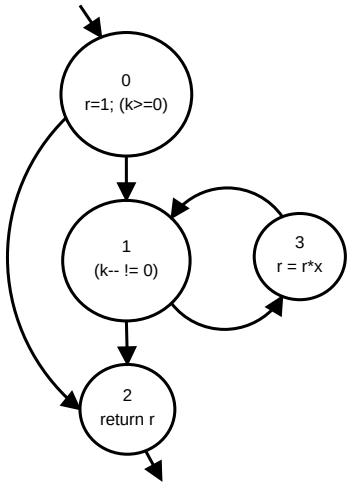
for some paths t and u.
□

Path coverage is the percentage of maximal paths through the CFG which are toured during the testing. Full path coverage obviously implies full branch coverage, but not the other way around, as have been demonstrated before.

As a side note, we can also express statement and branch coverage in terms of CFG. *Node coverage* (or 'block coverage') 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 maximal paths in the graph. Covering all of them is impossible. A reasonable alternative is to require something along the line of iterating every cycle at least once. But note that a loop may have branches inside, or another loop nested in it. To just requireing it to be iterated at least once sounds too simplistic in this situation (it does not feel adequate).

An *elementary* path in a CFG is a path where no node occurs more than once, except if it appears as the first and last one. In the latter case, the path forms a cycle. But such a cycle is also 'elementary' in the sense that it cannot contain another cycle. An elementary path is *prime* if it is not a subpath of another elementary path. The concept of 'prime path' is originally proposed by Amman and Offutt [1]. *Prime path coverage* is the percentage of the number of prime paths of the program which are toured during the testing.

For example, all paths (4) in the program `Q` are also prime paths. On the other hand, the program `power` has infinite number of paths, but it has only 5 prime paths:

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

The path $[0, 1, 2]$ corresponds to the execution where the loop immediately terminates (it does 0 iteration). The cycles $[1, 3, 1]$ and $[3, 1, 3]$ represent a whole class of executions where the loop does at least one iteration.

Note that an execution σ that tours one of the cycles ($[1, 3, 1]$ or $[3, 1, 3]$) will not tour $[0, 1, 2]$ (the 0-iteration case), is the latter cannot be a subpath of such a σ. So, $[0, 1, 2]$ really represents a different kind of executions.

However, arguably the two cycles $[1, 3, 1]$, and $[3, 1, 3]$ actually represent the same cycle, but the definition of prime path coverage treats them as different. Touring $[1, 3, 1]$ would require an execution that loops at least once, whereas touring $[3, 1, 3]$ requires at least two iteration. But then we can point out that any execution that tours the latter also tours the first. In fact, it will also tour $[3, 1, 2]$. Alternatively we can require to only cover a smallest set of prime paths instead:

Definition 1.2.2 : Smallest Set of Prime Paths
Let J be the set of all prime paths in G. A subset $I \subseteq J$ is a smallest set of prime paths if for any path $\tau_1 \in J/I$, there exists $\tau_2 \in I$ such that every execution path that tours τ_2 will also tour τ_1.
□

So, such a set can be ontained by dropping any prime path τ_1 if there is another prime path τ_2 such that any execution path that tours τ_2 will also tours τ_1[3].

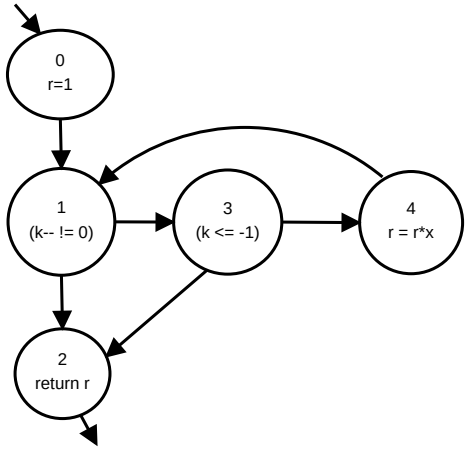
A smallest set of prime paths of `power` is:

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

At most three test-cases are needed to give full coverage on this set.

Let us now consider a more complicated loop. Cosider this slight variation of the program `power`:

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



A smallest set of prime paths of `powerz` is:

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

At most five test-cases are needed to fully cover them, though we can actually do it with just four.

The first path, $[0, 1, 2]$, can only be toured by an execution that does not enter the loop. The second, $[0, 1, 3, 2]$ can only be toured by an execution that enters the loop, and then immediately breaks.

The cycle $[4, 1, 3, 4]$ can be toured by any execution that iterates at least twice. The last two require that we should have one execution that iterates then terminates normally, and another that iterates and then terminates through the break.

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` $\not=$`0` on the program `powerz` shown before, then it is impossible to cover the path $[0, 1, 2]$. Such a path is called *unfeasible*.

Let us first consider two weaker variants of 'tour'. Let σ be a path. The induced edges-sequence of σ is:

$$[(\sigma_i, \sigma_{i+1}) \mid 0 \leq i < N]$$

where N is the path's length (which is the length of the list σ minus 1).

[3]But an objection to this alternative is if τ_2 turns out to be unfeasible. But in practice, it is not so likely that we have a loop that always terminate after just one iteration.

Definition 1.2.3 : Sidewalk
Let σ, τ be two paths, and s, t are their respective edge-sequences. The path τ is said to *sidewalk* σ, if all edges in s appear in t, and they appear in the same order.

Another way to say this is that s can be obtained from t by deleting some elements in t.
□

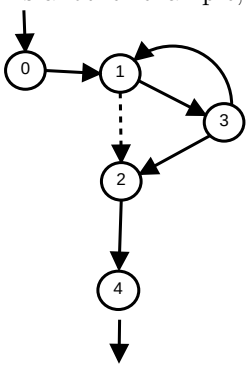
Definition 1.2.4 : Detour
An execution path τ is said to *detour* a path σ, if all nodes in σ appear in τ, and they appear in the same order.
□

Of course, if τ tours σ then it also sidewalks and detours σ. Sidewalking is stronger than detouring: if τ sidewalks σ then it also detours σ. The reverse directions are not necessarily true.

For example, the execution path $[0, 1, 3, 4, 1, 2]$ in `powerz` (it does one iteration, then terminates) does not tour $[0, 1, 2]$, but it still sidewalks the latter.

If *powerz* turns out to require $k > 0$ as pre-condition, then the path $[0, 1, 3, 2]$ is feasible. Thus, no actual execution path in its CFG can tour it. Actually, no actual execution path can even sidewalk it. However, the path $[0, 1, 3, 4, 1, 2]$, which is feasible, can still detour it ($[0, 1, 3, 2]$).

As another example, consider the CFG below:



Suppose that the cycle, if entered, it must be iterated at least once. Then the prime path $[0, 1, 3, 2, 4]$ cannot be toured. However, it can be sidewalked by the path $[0, 1, 3, 1, 3, 2, 4]$.

If the dashed edge $(1, 2)$ is unfeasible, then the prime path $[0, 1, 2, 4]$ cannot be toured, neither can it be sidewalked. However, the previous execution $[0, 1, 3, 1, 3, 2, 4]$ detours $[0, 1, 2, 4]$.

In general, when we have difficulty to tour a certain path σ, it could be (though we may not know for sure) that it is because the path is actually unfeasible. We can then at least check if it can be covered if we weaken our definition of touring (so, either sidewalking or detouring). Of course, we are still unable to tour σ, but in my oppinion knowing that it can still be sidewalked or detoured is still useful information.

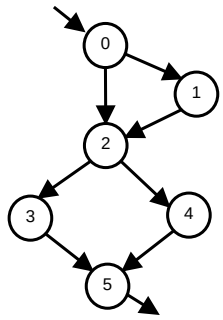
Unrelated with the feasibility issue, prime path coverage by detour is strictly more powerful than node coverage, and prime path coverage by sidewalking is strictly more powerful than edge coverage. Consider again the above example.

- A single execution $[0, 1, 3, 2, 4]$ is enough to give a full node coverage, but it will not detour the prime path $[3, 1, 3]$.
- Two executions paths $[0, 1, 2, 4]$ (no iteration) and $[0, 1, 3, 1, 3, 2, 4]$ (one iteration) gives a full branch coverage. However, the prime-path $[3, 1, 2, 4]$ are not side-walked by either of those execution paths.

1.2.2 Linearly Independent Paths

The number of prime paths is exponential. So, if you have a program with e.g. 10 if-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.

1.3 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*.

1.3.1 Classification tree for black box testing

A practical approach you can use for black box testing is by using classification trees [8]. Consider a hypothetical program to register grades for students. The program has this header:

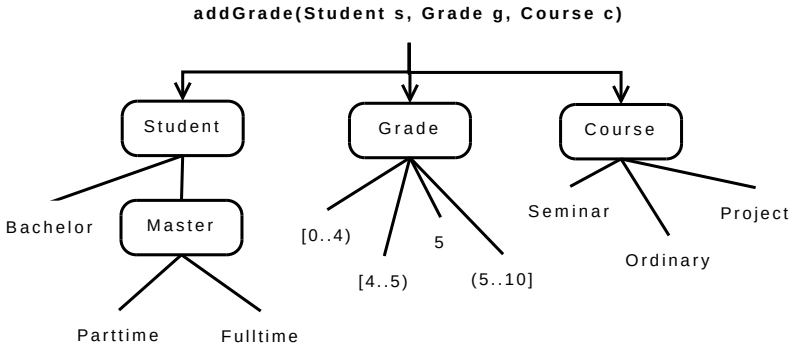
```
addGrade(Student s, Grade g, Course c)
```

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:

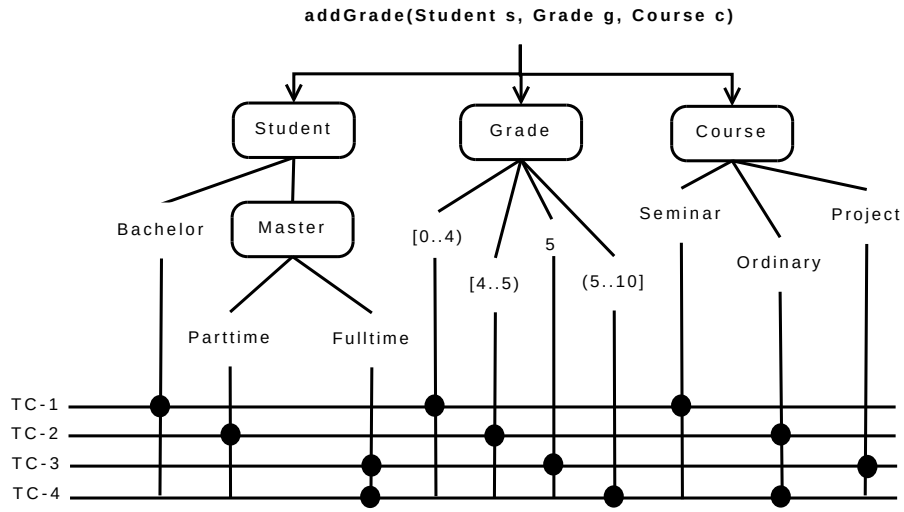


The tree above is called *classification tree* (CT). The boxes and the leaves represent domains or subdomains. In CT jargon they are called 'classes'. The top level classes are called *categories* (above: *Student*, *Grade*, and *Course*), each repreenting a parameter of `addGrade`.

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 e.g. such that every class would be tested at least once to give full CT class coverage, or such that every pair of leaf-classes are tested at least once.

There is also a tool called Classification Tree Editor (CTE) from `systematic-testing.com` that allows you to generate such sets of combinations of leaf-classes, or to specify a more advanced set by imposing constraints on the combinations you want, or to even 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).

1.4 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. This means modifications are introduced. Furthermore, users may request changes in features or functionalities, or new ones to be added. This again triggers modifications in the software.

Supposed you have introduced a set of modifications. We would want to test that at least they do not break the functionalities that are supposed to be preserved. The pragmatic way of doing this is by re-running the test suite of the previous version. This is called *regression testing*. If an error is found, this does not necessarily mean that it is a bug in the new version. It could also be that it is part of the intended behavior of the new version which was not accepted or even present in the old version. But at least, it warns us that something needs to be investigated. Also, when no error is found in regression testing, we should keep in mind that any new feature would not be covered in such a test. So, additional test-cases may be needed.

When the SUT is large, its test-suite T is typically very large. Re-running the whole can take hours, or even days. Since modifications are typically limitted to just some places in the SUT, it is likely that most test-cases in T are actually irrelevant for testing the modifications. But how can we figure out which of them are should be included (without having to first execute the whole T to find that out)? Unfortunately, in general this is an undecidable problem due to the halting problem it implies. So, we will consider a slightly different problem instead.

1.4.1 Test cases selection

Let's first define some concepts. For simplicity, let's assume the SUT to be deterministic. A test-case t is asbtractly described by a pair (i, o) where i is the input (or inputs) for the SUT, and o is the resulting result of the test-case, which is either success or fail. For simplicity, let us assume that crash and non-termination to be detectable, and will also result in fail.

Let P be the SUT, and T be test suite we used to test it. We assume that P has passed T. That is, none of the test-case in T failed. Let P' be the new version of P after some modifications. Some of the test-cases in T may no longer be syntactically correct for testing P', e.g. when compiling them simply fails. These test-cases are *obsolete*. Obviously, we should exclude them from regression.

What we want is to select a subset $S \subseteq T$ (without re-executing T to do so) to do regression testing on P'. With a smaller S we hope to reduce the test execution time[4]. But we should also include the cost of the selection itself. The total time we spend on selecting S and executing it should not exceed the time needed to simply re-execute the whole T (minus the obsolete test-cases).

A test-case $t \in T$ is *fault revealing* (with respect to P') if it fails on P'. A selection algorithm is *safe* if the resulting S is guaranteed to include all fault revealing test-cases. It is *minimal* if it only contains fault revealing test-cases. Finding the minimal S is in general, again, an undecidable problem.

Suppose we can instrument the SUT, so that when we execute a test-case we knows which 'locations' in the SUT we visited by the execution. So, for each test-case in T, we have this data when we used it to test P. For a test-case t to be fault revealing when used to to test P', it must pass a *modified part*. This is a part which either new in P', or it was in P but deleted in P', or it is present in both but is at least textually different in both version (it has been modified in P').

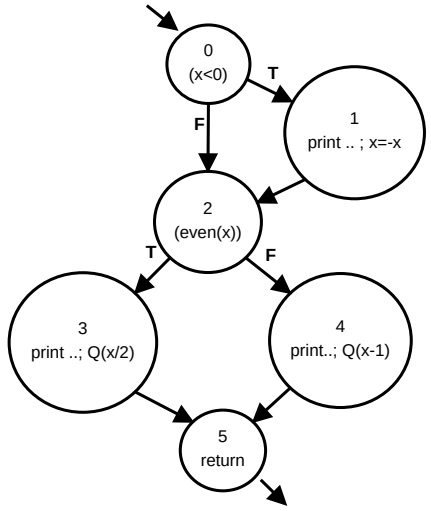
[4]But of course if the size of S_1 is smaller than the size of S_2, it does not necessarily mean that S_1 will execute faster than S_2.

A test-case that does not pass any modified part will have exactly the same result in P', and thus cannot be fault revealing.

A test-case that passes a modified part, is called *modification revealing*. But note that when the code is textually modified, the semantic can still be the same. So, a modification revealing test-case is not necessarily fault revealing. In other words, the first concept is weaker than the latter.

Selecting all modification revealing test-cases is safe, but this is less trivial than it may seem due to deletion and insertion of new statements. Let's first introduce some concepts.

We will first tweak our representation of 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 outgoing arrows. We will label 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:



Two nodes u, v are syntactically equivalent, denoted by $u \equiv v$, if the statements they represent are textually the same.

Let G_1 and G_2 be the CFGs of P and P'. Consider a test-case t. Suppose when executed on P and P' it traverses the path τ_1 respectively τ_2. If t is modification revealing, these paths must pass some nodes that are syntactically different. In particular, there must a pair of nodes $v_1 \in G_1$ and $v_2 \in G_2$ which the traversed paths differ for the first time from each other. This implies that τ_1 and τ_2 must be of this form:

$$\tau_1 \;=\; \sigma +\!\!+ [v_1] +\!\!+ v_1$$

$$\tau_2 \;=\; \sigma +\!\!+ [v_2] +\!\!+ v_2$$

where $v_1 \not\equiv v_2$, and σ is a common prefix of τ_1 and τ_2. Furthermore, branch-decissions made along σ and its next node must have been identical in both executions. That is, the list of labels over the edges in $\sigma +\!\!+ [v_1]$ (in G_1) and the list of labels over the edges in $\sigma +\!\!+ [v_2]$ (in G_2) must be the same.

So, what we can do is to quantify over all paths $\tau_1 \in G_1$, and to find the pair (v_1, v_2) as above, which is the first pair of nodes where it would differ than its counterpart $\tau_2 \in G_2$. Although there may be infinitely many paths in G_1 and G_2, there will only be finite number of such (v_1, v_2).

Figure 1.1 shows an algorithm by Rothermel and Harrold [13]. The inputs are the CFGs and the test suite T (we assume that obsolete test-cases have been filtered out). The goal of the algorithm is to return the set of all modification revealing test-cases.

```
select(G1, G2, T) {
   modified := ∅
   seen     := ∅
   compare(root(G1), root(G2))
   return {t | n∈modified, t∈T, t visited n}

   where
   compare(n1, n2) {
     seen := seen ∪ {(n1, n2)}
     if (n1≢n2) {
          modified := modified ∪ {n1}
          return
     }
     else {
         for(o1∈next(G1)) {
             let o2∈G2 such that label(n1→o1) = label(n2→o2)
             if ((o1, o2)∉seen) compare(o1, o2)

         }
     }
   }
}
```

Figure 1.1: Rothermel and Harrold selection algorithm.

1.5 Literature

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

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:

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

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 called pre-condition, and Q post-condition. The specification 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.

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. $\mathsf{wp}\ (\ \mathtt{if}\ g\ \mathtt{then}\ S\ \mathtt{else}\ T)\ Q \quad = \quad (g \Rightarrow (\mathsf{wp}\ S\ Q))\ \wedge\ (\neg g \Rightarrow (\mathsf{wp}\ T\ Q))$

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. Recall also that we remarked that the SEQ Rule (2.1.2) to handle $S_1; S_2$ is not so nice because we have to come up some intermediate predicate P' ourselves. However, if the weakest pre-condition of $S_1; S_2$ then it is sufficient to prove $P\ \Rightarrow\ \mathsf{wp}\ (S_1; S_2)\ Q$ instead –here, we do not need to come up with such a P'.

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

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

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:

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

Answer: using the pre-condition $\mathtt{i}{\geq}\mathtt{0}$ 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:

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

Answer: using this as invariant:

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

Prove the conditions yourself.

Example

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

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

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

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

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

Answer: using this as invariant:

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

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 this case 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:

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

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:

$\{* \texttt{r}{\geq}0 \ \wedge \ \texttt{b}{\geq}0 \ *\}$

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

$\{* \texttt{r}{=}0 \ \wedge \texttt{b}{=}0 \ *\}$

The `if` above is non-deterministic. So, if the specification is valid we will end up with both no more candy in the bag ($\texttt{r}{=}0 \wedge \texttt{b}{=}0$), 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 : $\texttt{r}{\geq}0 \ \wedge \ \texttt{b}{\geq}0$

m : $\texttt{r} + 2\texttt{b}$

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 $\Sigma \to val$, 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:

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

Conversely, if we know the set, it defines the corresponding predicate. Therefore I will treat predicates and the sets of states that they induce as interchangeable.

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

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

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 `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:

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

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

⟨*type*⟩ ::= ⟨*simple-type*⟩ | ⟨*record-type*⟩ | ⟨*array-type*⟩ | ⟨*type-name*⟩

⟨*simple-type*⟩ ::= `()` | `bool` | `int` | `char` | `string`

⟨*record-type*⟩ ::= `record` '{' ⟨*field-def*⟩ (';' ⟨*field-def*⟩)* '}'

⟨*field-def*⟩ ::= ⟨*field-name*⟩ (',' ⟨*field-name*⟩)* ':' ⟨*simple-type*⟩

⟨*array-type*⟩ ::= (⟨*simple-type*⟩ | ⟨*record-type*⟩)`[]`+

⟨*expr*⟩ ::= ⟨*constant*⟩
| ⟨*variable-name*⟩
| ⟨*array-element*⟩
| ⟨*record-element*⟩
| ¬⟨*expr*⟩
| ⟨*expr*⟩ ⟨*binary-operator*⟩ ⟨*expr*⟩
| '(' ⟨*expr*⟩ ')'

⟨*array-element*⟩ ::= ⟨*variable-name*⟩ ('[' ⟨*expr*⟩ ']')+

⟨*record-element*⟩ ::= ⟨*expr*⟩ '.' ⟨*field-name*⟩

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)

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

Rule 2.8.3 : Contradiction

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

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)

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

Rule 2.8.7 : ∨ Elimination

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

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

$$\frac{P \Rightarrow Q \quad \neg P \Rightarrow Q}{Q} \qquad \frac{P_1 \vee P_2 \quad P_1 \Rightarrow Q \quad P_2 \Rightarrow Q}{Q}$$

Rule 2.8.10 : Specialization (∀-Elimination)

$$\frac{P\ e \quad (\forall i : P\ i : Q\ i)}{Q\ e} \qquad \frac{(\forall i : P\ i : Q\ i)}{P\ e\ \Rightarrow\ Q\ e}$$

Rule 2.8.11 : ∃ Introduction

$$\frac{P\ e}{(\exists i :: P\ i)} \qquad \frac{P\ e \quad Q\ e}{(\exists i : P\ i : Q\ i)}$$

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 $\forall$

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

Theorem 2.8.19 : Negate $\exists$

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

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.

Note that an if-then-else structure can be encoded as follows:

$$\texttt{if } g \texttt{ then } S_1 \texttt{ else } S_2 \quad = \quad \{\texttt{assume } g \ ; \ S_1\} \ [] \ \{\texttt{assume } \neg g \ ; \ S_2\}$$

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 $u = \texttt{new Point}()$ 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:

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

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. $\mathsf{wlp}\ \texttt{skip}\ (N, -, -) \quad = \quad N$
2. $\mathsf{wlp}\ (x = e)\ (N, -, -) \quad = \quad N[e/x]$
3. $\mathsf{wlp}\ \texttt{raise}\ (-, X, -) \quad = \quad X$
4. $\mathsf{wlp}\ (\texttt{assert}\ P)\ (N, -, X) \quad = \quad (P \wedge N) \vee (\neg P \wedge X)$
5. $\mathsf{wlp}\ (\texttt{assume}\ P)\ (N, -, -) \quad = \quad P \Rightarrow N$
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.
   $\mathsf{wlp}\ (\texttt{var}\ x\ \texttt{in}\ C\ \texttt{end})\ Q \quad = \quad (\forall x :: \mathsf{wlp}\ C\ (N, X, W))$

2.10.4 Handling Loops

As you notice (our) GCL does not have a loop, but it does not mean that we cannot handle them. Let us first conside be a loop which is annotated with an invariant:

$\texttt{while}\ g\ (inv\ I)\ \texttt{do}\ S$

We convert this to:

$\texttt{assert}\ I\ ;$
$\texttt{if}\ g\ \texttt{then}\ \{S;\ \texttt{assert}\ I;\ \texttt{assume false}\}\ \texttt{else skip}$

This effectively encode the conditions in Rule 2.3.1 for verifying loops. Note that the above translation forces you to prove the following:

1. That I is established just before we enter the loop.
2. That S satisfies $\{* \ I \wedge g \ *\}\ S\ \{* \ I \ *\}$.
3. That any post-condition Q we set for the loop satisfies $I \wedge \neg g \Rightarrow Q$.

If no invariant is specified, or if I is very weak, we can alternatively choose to 'unroll' the loop up to k-times. E.g. to unroll it 0 and 1 times, we translate it to (if no invariant is given, then use `true` as I):

$\texttt{assert}\ I\ ;$
$\texttt{if}\ g\ \texttt{then}\ \{S;\ \texttt{assert}\ I;\ \texttt{assume}\ \neg g\}\ \texttt{else skip}$

Note that we now use $\texttt{assume}\ \neg g$ rather than `assumefalse`. The above will verify the correctness of the loop when it does 0 or 1 iterations. When it turns out to take more iterations, we cannot say anything about its correctness. In other words, the verification is thus incomplete.

2.11 Literature

- E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976
- I.S.W.B. Prasetya. *Introduction to Progamming Logic.* Dept. of Inf. & Comp. Sciences, Utrecht Univ., 2012. Lecture Notes of the course Software Testing and Verification, online available at `www.cs.uu.nl/docs/vakken/pc`
- K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. Technical Report SRC Technical Note 1999-002, Compaq, 1999. available online at `http://research.microsoft.com/en-us/um/people/leino/papers/SRC-1999-002.pdf`

Chapter 3

Model Checking

3.1 Automata

Figure 3.1 shows some an example of a *finite state automaton* (FSA), also known as finite state machine (FSM). Such an automaton is often used to (abstractly) model a program. The nodes in the 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. As the name suggests, these automata have finite number of states.

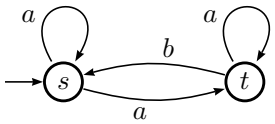


Figure 3.1: A finite state automaton

Definition 3.1.1 : Automaton
Let us define an *automaton* M as 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.

It is an FSA is the set S is finite. □

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 also just call this sequence of states '*execution*'. It should be clear from the context which one is meant.

In the literature you will find many variations of the above definition; figure 3.2 shows some such variations.

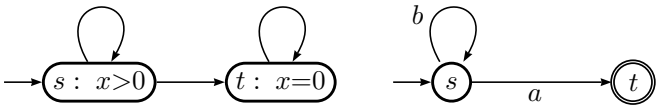


Figure 3.2: Finite state automata

The one on the left has no label/decoration on the arrows, but has labels on the states instead. The one on the right has a so-called 'accepting state', marked with the double-lined border.

Let's call a state with no successor a *terminal state*: no further execution is possible from such a state. Accepting and terminal states are not the same concept. Accepting states are often used to express certain goals; they may coincide with terminal states, but not necessarily so.

Depending on what aspects you need to capture in your FSAs you may or may not need such extensions. 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 (as the right automaton in Figure 3.2), we may only want to consider executions that end in an accepting states.

The right automaton in Figure 3.2 is *deterministic*, because from any state u, the action α uniquely determines what the next state is (in other words, $R(u, \alpha)$ has at most one state). On the other hand, the automaton in Figure 3.1 and the left automaton in Figure 3.2 are *non-deterministic*.

A real program P operates on variables. The program's *concrete state* is determined by the values of its variables. When we model the program with an automaton M, we basically map P's concrete states to M's states. This mapping can be one-to-one. That is, each P's concrete state is mapped to a unique state in M. Such an M will have a large number of states (though this number may still be finite).

We can also choose to model more abstractly, e.g. 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 automaton called *Kripke structure*.

Definition 3.1.2 : Kripke Structure
A Kripke structure K is a finite state automaton, described by a tuple $(S, I, R, Prop, V)$ where:

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

  The notation $Pow(S)$ 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.
- $V : S \to Pow(Prop)$ 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 $p \notin V(s)$ then p does *not* hold in s.

  The labelling is assumed to be consistent. That is, for every state s, the conjunction:

$$\bigwedge_{p \in V(s)} p \quad \wedge \quad \bigwedge_{q \notin V(s)} \neg q$$

  should be satisfiable (in other words, the conjuction should not be empty/contradictive).

□

Example:

We have above two states, named s and t. We have two propositions, $isOdd(x)$ and $x>0$. We can see above that $isOdd(x)$ holds in s, but $x>0$ does not, whereas in state t both propositions hold. Note that these propositions are 'independent' according to the definition above.

Formally, the R of the above Kripke structure is:

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

And the V:

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

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.

Checking whether a proposition p holds in some or all states of a give Krike structure is straight forward: we can just iterate over all states and check if p decorates them. We have afterall only finite number of states.

Checking that a certain property (e.g. that p will occur infinitely many often) hold on all executions is different. Although we have finite number of states, the number of executions the automaton generates may be infinite (as the automa Figure 3.1). So, simply enumerating the executions will not work. Furthermore, some executions may be infinite. We will returnt to this issue later.

3.2 Some Basic Operations on Automata

In this section we will assume the concept of automaton as in Definition 3.1.1.

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:

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

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.

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

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:

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

- 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 $\phi \wedge \psi$:

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

Some examples of properties you can express with LTL:

- $\Box(p \to \Diamond q)$: whenever p holds, then eventually q will hold.

- p **U** (q **U** r): we start with a possibly empty duration where p holds constantly, followed immediately by a (also possibly empty) duration where q holds constantly, which will be ended by a point where r holds.
- $\Diamond\Box p$: eventually we will come to p, afterwhich we will remain in p (eventually, we stabilize into 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[1]. 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 $M = (S, s_0, R, Prop, V)$, 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* τ of a Kripke structure is as defined generically in Section 3.2, except that the arrows of a Kripke structure has no label. This τ induces a sequence of propositions that hold along the execution, which is the sequence:

$$V(\tau_0), V(\tau_1), ...$$

We will call the latter *abstract execution*, or just *execution* if it is obvious from the context which one is meant.

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:

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

[1] In literature this is also called: extending executions with 'stuttering'.

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:

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

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
A 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 there is no labelling function, and that now we have F.

- 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.

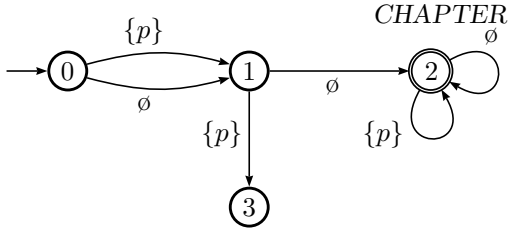
□

A Buchi automaton M only accepts infinite sentences. A infinite execution τ of M is accepting if it passes through F infinitely many times[2]. 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 is indeed a bit wierd because its alphabet Σ must be $pow(Prop)$ (and *not* $Prop$).

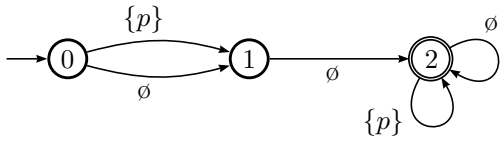
For example, the Buchi automaton below fully describes the abstract executions of the formula $\mathbf{X}\ \neg p$, over $Prop = \{p\}$:

[2]It does not have to pass *every* state in F inifinitely many times. Since F is finite, τ just needs to pass at least one state in F inifinitely many times.



Note that the alphabet Σ of the automaton is $pow(Prop)$. So, in this case $\Sigma = \{\emptyset, \{p\}\}$, which means that we have *two* possible symbols to label each arrow: "$\emptyset$" or "{p}". The second one represents the case when p holds, whereas the first one represents the case when p does not hold.

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 $s \xrightarrow{a} t$.

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

  We can go from s to t if the propositions p and q hold.

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

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

  We can go from s to t if the proposition p and q both do not hold.

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

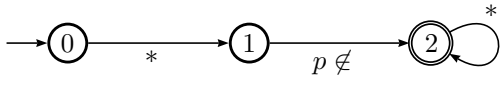
- $s \xrightarrow{C_1, C_2, \ldots} t$

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

  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 $s \xrightarrow{a} t$.

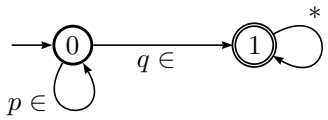
  E.g. we can thus write $s \xrightarrow{p\in, q\notin} t$

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

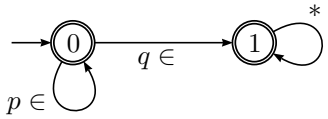


Here are few more examples:

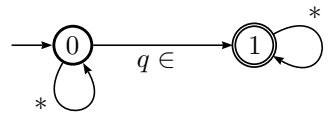
- A Buchi automaton that describes the formula p **U** q:



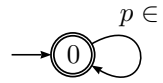
- A Buchi automaton that describes the formula p **W** q; notice that we have two accepting states:



- A Buchi automaton that describes the formula $\diamond q$:



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



3.4.1 Generalized Buchi Automata

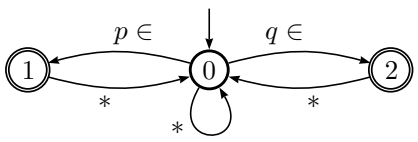
To represents some formulas, in particular conjunctions, it is convenient to use a *generalized Buchi automaton*. It is a Buchi automaton with multiple sets of accepting states. Note that we are talking about set of sets here. An ordinary Buchi automaton may have one, or multiple, accepting states. They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

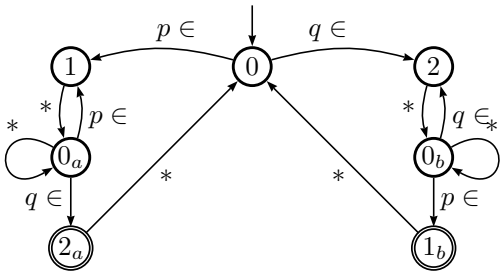
An execution is accepting by the GBA M if it passes through *each* member of $\mathcal{F}$ infinitely many times.

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



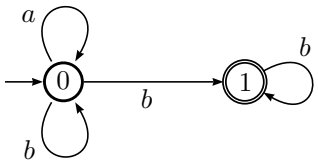
The $\mathcal{F}$ is $\{\{1\}, \{2\}\}$. So, we must visit state-1 infinitely many times, *and* also visit state-2 infinitely many times. Note that this is different that saying, as in the ordinary Buchi automaton, that $\{1, 2\}$ is our set of accepting states, for which passing state-1 infinitely many 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). Below is an ordinary Buchi automaton that accepts the same language as the one above. It has a single set of accepting states, which is $F = \{2_a, 1_b\}$.



3.4.2 Non-deterministic vs Deterministic Buchi Automata

Whereas a standard non-deterministic FSA can be converted into an equivalent deterministic FSA, this does not in general apply to Buchi automata. For example, consider the Buchi automaton below:



Sentences accepted by this automaton has the form $s{+\!\!+}[b, b, b, ...]$, where s is any finite prefix over a and b. No deterministic Buchi automaton can generate these sentences.

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. This will be discussed in the lecture.

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 produced by both M and $B_{\neg\phi}$. If one of these sentences, say σ, is also an *accepted* sentence 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. Formally, 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. To 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 each 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 or sets of the combined automaton C is just the same as those 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 G has nodes, connected by arrows. We can describe it with a tuple $(S, I, next)$ where S is its set of nodes, I is its set of roots (initial nodes), and $next : S \to pow(S)$ 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. A Buchi automaton $C = (S, I, F, R, \Sigma)$ induces a graph $G = (S, I, next)$ where $next(u) = (\cup a :: R(u, a))$.

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 state.

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 explores a given graph starting from a given node r. It follows the arrows as deep as possible, and then backtracks. When it terminates, it would have visited all nodes reachable from r. The algorithm is shown below.

```
explore(G) {
  visited := ∅ ;
  for(r∈root(G)) DFS(r)
    where
    DFS(u) {
      if (u∈visited) return ;  -- (*)
      visited := visited ∪ {u}  ;
      for (s : G.next(u)) DFS(s) ;
    }
}
```

Importantly, we maintain the set of the nodes we have seen (the variable `visited`). In this way the algorithm avoids visiting the same node twice, and thus avoiding to go into a cycle. At the end, `visited` will contain all nodes reachable from the root. Note that every node and edge in the graph are processed at most once. Consequently, its (worst) time complexity is $O(|S| + E)$, where E is the number of edges in G.

Notice that when the algorithm encounters a node u which is already in `visited` (in the step marked with (*) above), it implies that we have found a cycle, of which u is an element of. If you recall Theorem 3.5.2 we are indeed interested in finding cycles, but not just any cycle. We want to find cycles through an accepting state, which are also reachable from the root. For this purpose we modify the DFS algorithm to the double DFS algorithm shown below. The set of nodes in G that correspond to C's accepting states is denoted by acceptStates(G).

```
checkEmpty(G) {
  visited₁ := ∅ ;
  for(r∈root(G)) DFS(r)

    where
    DFS1(u) {
      if (u∈visited₁) return ;
      visited₁ := visited₁ ∪ {u} ;
      for (s : G.next(u)) {
         if (u∈acceptStates(G)) {
              astate := u ;
              visited₂ := ∅ ; DFS2(s)
              where
              DFS2(v) {
                 if (v = astate) throw CycleFound ;  -- (*) cycle!
                 if (v∈visited₂) return ;
                 visited₂ := visited₂ ∪ {v}
                 for (s : G.next(v)) DFS2(s) ;
              }
         }
      }

      DFS1(s) ;
    }
}
```

When it throws `CycleFound` in the step marked by (*), we can conclude that:

1. there is a cycle in G that contains an accepting state v.
2. v is reachable from the root.

So, by Theorem 3.5.2 this proves that the Buchi automaton C is not empty. What is even better, we can easily extend DFS so that it also gives us the path that leads to the cycle we

```
checkEmpty(G) {
  visited1 := ∅ ;
  for(r∈root(G)) DFS(r)

    where
    DFS1(u, σ) {
      if (u∈visited1) return ;
      visited1 := visited1 ∪ {u} ;
      for (s : G.next(u)) {

          if (u∈acceptStates(G)) {
              astate := u ;
              visited2 := ∅ ; DFS2(s, [u])
              where
              DFS2(v, τ) {
                  if (v = astate) throw CycleFound(σ++τ++[v]) ;  -- (*) cycle!
                  if (v∈visited2) return ;
                  visited2 := visited2 ∪ {v}
                  for (s : G.next(v)) DFS2(s, τ++[v]) ;
              }
          }

          DFS1(s, σ++[u]) ;
      }
    }
}
```

Figure 3.3: Double DFS model-checking algorithm.

found above. This allows us to show a concrete 'witness' of proving the non-emptiness of C. In terms of the orginal verification problem posed in Theorem 3.5.1, where we ask the question whether the LTL formula ϕ is valid on M, this path/witness correspond the an execution in M that demonstrates a violation of ϕ. In otherwords, it concretely shows you an execution producing the error, which is of course a very useful information for debugging.

Figure 3.3 shows a version of the above double DFS algorithm, this time extended so that we also keep track of the path that leads to the node we are inspecting. More precisely, we maintain these invariant:

In DFS1: σ is a path from the root up to (but not including) u.

In DSF2: $\sigma +\!\!+ \tau$ is a path from the root up to (but not including) v.

When an accepting cycle is found (the step marked with (*) in Figure 3.3), we also report the full path to the cycle, and the cycle itself, which is:

$$\sigma +\!\!+ \tau +\!\!+ [v]$$

Let the 'size' of an automaton K, denoted by $|K|$, be expressed as the sum of the number of its states and the number of its edges. When converting an LTL formula ϕ to a Buchi automaton we may end up with an automaton B whose size is exponential with respect to the size of ϕ (denoted by $|\phi|$), which is defined as the number basic terms it contains.

The intersection $M \cap B$ can be in the worst case of size $|K| * |B|$. It follows that our DFS algorithm is $O(K * 2^{|\phi|})$ where $K = |S| + E$, which can be seen as expressing the size of your target Krikpe structure, and $|\phi$ is the size of the LTL formula you are verifying.

3.5.2 By-passing the Intersection

The algorithm in Figure 3.3 assumes that G is the graph representing the intersection automaton $Buchi(M) \cap B_{\neg\phi}$. However, we do not actually need to construct this intersection first. This would save us memory and CPU time. Effectively, the algorithm only needs the following information from G: $root(G)$, $G.next(u)$, and $\mathsf{acceptState}(G)$. These can be directly expressed in terms of M and $B_{\neg\phi}$. We will represents the nodes in G by pairs (s_1, s_2) where s_1 is a state in M and s_2 is a state in $B_{\neg\phi}$.

1. $(r_1, r_2) \in root(G)$ can be equivalently checked by $r_1 \in init(Buchi(M)) \ \wedge \ r_2 \in init(B_{\neg\phi})$
2. $(s_1, s_2) \in next((u_1, u_2))$ can be equivalently checked by checking the existence a transition $u_1 \xrightarrow{A} s_1$ in $Buchi(M)$ and a transition $u_2 \xrightarrow{A} s_2$. Note that labels of these transition must be the same.
3. $(u_1, u_2) \in \mathsf{acceptState}(G)$ can be equivalently checked by checking that u_1 is an accepting state in $B_{\neg\phi}$.

So, we can replace the above checks in the algorithm in Figure 3.3 with their equivalent counterparts.

3.5.3 Model Checking on Concrete States

Even with the improvement proposed above our model checking algorithm still assumes that M is a Kripke structure. If we have a real program P, usually it is not expressed as a Kripke structure. Can we then systematically construct a Kripke structure out of it?

First, note that we have defined Kripke structures to have finite number of states. Actually, this was because the model checking algorithm really requires you to have finite number of states. If P has infinite number of states, in theory the corresponding Kripke structure M can still be finite, depending on the chosen abstraction $Prop$. But to automatically construct M will still be problematic as we will need to quantify over P's concrete states, whose number is infinite.

If P has finite number of states (or, you only consider a subset of the original program so that you only have to deal with finite number of states), we can construct M e.g. as shown below:

1. Let's introduce two sets S, R, intialized to empty. S will be a set of state, and R a set of arrows/transitions between the states.
2. For every initial state $s_0 \in init(P)$, we explore s_0. To explore a state s we do:
   (a) if we have seen s before, we won't explore it again. Else we add it to S.
   (b) We check the code of P to see what's the next 'atomic' statement[3] to execute. If there is none, we are done exploring s.
   If there is only one atomic statment a possible, we execute it. Suppose t is the resulting state. We add t to S, and the arrow (s, a, t) to R. Then we explore t.
   If there are multiple atomic statements possible on s, we do as above for every choice b that we have.

This procedure terminates, and at the end $(init(P), S, R)$ defines an FSA K that is equivalent to P. If $Prop$ is given, it is straight forward to reduce K to a Kripke structure M. What is not so nice here is that intermediate FSA K is typically very big. Now, if we look again at the improvement discussed in Subsection 3.5.2, the only part of model checking that actually requires information from M is this part:

> Check the existence a pair of transitions, $u_1 \xrightarrow{A} s_1$ in $Buchi(M)$, and $u_2 \xrightarrow{A} s_2$ in $B_{\neg\phi}$, labelled by the same A.

[3] An atomic statement is a statement whose execution cannot be interrupted.

Recall that the orginal place in the algorithm in Figure 3.3 where the above is used is in this quantification:

for $((s_1, s_2) \in next(u_1, u_2))$ *body*

Using the above improvement in Subsection 3.5.2 it is then replaced by:

forall (a_1 is an outgoing arrow in $Buchi(M)$ from u_1) `-- *1`
  forall (a_2 is an outgoing arrow *in* $B_{\neg\phi}$ from u_2) {
    if ($label(a_1) \neq label(a_2)$) **continue** ; `-- *2`
    s_1 := $destination(a_1)$; `-- *3`
    s_2 := $destination(a_2)$;
    body
  }
}

Notice that only the parts marked by * actually requires information from $Buchi(M)$. The first one can be equivalently done by quantifying over all atomic statements α_1 of P that can be executed on the the current state u_1. However now, we no longer keep track of the states of M, but rather the concrete states of P.

In *2 we have to check whether the labels of a_1 and a_2 are the same. However, now we have α_1 instead of a_1, and being a statement of a real program, it does not have a label as a Buchi automaton would have. But actually the purpose of this checking is to see if all propositions p in $label(a_2)$ hold in the current state u_1. Since now u_1 is a concrete state, checking this can be done simply by evaluating p on u_1.

Finally, in *3 we need to know the next state of α_1, which we can do simply by executing it on u_1.

To summarize, the above code fragment can thus be replaced with the following:

forall (α_1 is an atomic statement in P that can be executed on u_1)
  forall (a_2 is an outgoing arrow *in* $B_{\neg\phi}$ from u_2) {
    if (there is a $p \in label(a_2)$ that does **not** hold on u_1) **continue** ;
    s_1 := execute a_1 on u_1 ;
    s_2 := $destination(a_2)$;
    body
  }
}

With this modification, the model checking algorithm is run directly on P. It does not construct any intermediate FSA. This approach is also called *on-the-fly* model checking. Notice that the 'explore' algorithm presented before (for constructing P's FSA) is actually also an instance of DFS. We can thus see the inner DFS in the model checking algorithm as being superimposed on the exploration DFS. From this perspective, it is as if we do model checking on-the-fly as we explore P.

Keep in mind that the above procedure of model checking directly on P presumes P to have finite number of states. For example, P can be a model expressed in Promela, which is the modelling language used by the model checker SPIN. Promela has the familiar style of a structured programming language, though it provide only a small set of programming constructs. In any case, models written in Promela always have finite number of states. On the other hand, if P is e.g. a Java program, then you will need to somehow constrain it to a finite subset of its full behavior. The result of your verification will indeed be incomplete, but with respect to the space you are checking you will still be complete.

3.6 Literature

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

- Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003

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 rooted in an initial state of K, and 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 trees that starts in K's innitial states are of course a bit special, since this represents K's all possible full executions.

The syntax of CTL:

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

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 $\mathbf{EU}$, 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:

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

where $init(K)$ denotes the set of K's innitial states.

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. $\mathbf{EX}\ \phi$ 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:

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

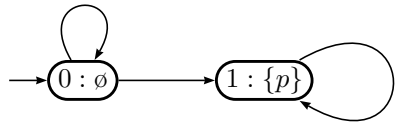
4.1.1 CTL vs LTL

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 $Prop = \{p\}$.

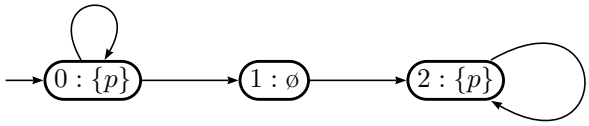


The CTL property $\mathbf{EF}\ p$ holds on this automaton, which means that there is one execution on which the proposition p will eventually hold.

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:



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 **AF** (**AG** p). But this requires that all execution paths from state-0 satisfies the **F**(**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 $tree(\pi_i) \models$ **AG** p. But $\pi = 0, 0, 0, ...$ So, its i-th state is just state-0, and **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:

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

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

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

In CTL* we can express e.g. **E**(p **U** (p **U** q)). Furthermore, 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, ϕ **U** ψ holds if ψ holds immediately, or if ϕ holds now the ϕ **U** ψ 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

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

□

4.1.4 CTL Model Checking

Consider a Krikpe structure $K = (S, I, R, Prop, V)$. 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:

$$K \models \phi \quad = \quad s_0 \in W_\phi, \text{for all initial state } s_0 \tag{4.7}$$

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:

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

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

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

   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:

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

   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):

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

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

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

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, more abstractly the above equation has this form:

$$Z \quad = \quad W_\psi \;\cup\; (W_\phi \;\cap\; f(Z)) \tag{4.8}$$

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:

$$Z_0 \;=\; \emptyset$$

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

$$Z_{i+1} \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; f(Z_i))$$

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

Obviously, if the above loop terminates, it will terminate with a solution of (4.8). Because of the way f is defined, you can actually prove that

$$Z_i \;\subseteq\; Z_{i+1}$$

Because K is a finite state automaton (S is finite), we cannot indefinitely grow Z_i. So, the loop must terminate. It can also be proven that it terminates with the smallest solution.

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:

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

Or:

$$Z \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; g(Z))$$

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:

$$Z_0 \;=\; \emptyset$$

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

$$Z_{i+1} \;=\; W_\psi \;\cup\; (W_\phi \;\cap\; g(Z_1))$$

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

Optimizing the fix-point iterations

The drawback of the above ways of calculating $W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)}$ and $W_{\mathbf{E}(\phi\ \mathbf{U}\ \psi)}$ is that states that have been added in the iteration k will be inspected and added again at every iterations after k. We can improve the the calculation of $W_{\mathbf{A}(\phi\ \mathbf{U}\ \psi)}$ as follows:

1. Start with these:

$$\begin{array}{lcl} Z_0 & = & \emptyset \\ Z_1 & = & W_\psi \end{array}$$

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

$$Z_{i+2} \ = \ Z_{i+1} \ \cup \ (W_\phi \ \cap \ f(Z_{i+1}))$$

where $f(Z)$ is defined as before, namely $f(Z) = \{s \mid s \in S \text{ and } R(s) \subseteq Z\}$. But let's do some calculation:

$$\begin{array}{ll} & Z_{i+1} \ \cup \ (W_\phi \ \cap \ f(Z_{i+1})) \\ = & \\ & Z_{i+1} \ \cup \ (W_\phi \ \cap \ f((Z_{i+1}/Z_i) \cup Z_i)) \\ = & // \text{ you can show that } f \text{ distributes over } \cup \\ & Z_{i+1} \ \cup \ (W_\phi \ \cap \ (f(Z_{i+1}/Z_i) \cup f(Z_i))) \\ = & // \text{ notice that: } W_\phi \ \cap f(Z_i) \ \subseteq \ Z_{i+1} \\ & Z_{i+1} \ \cup \ (W_\phi \ \cap \ f(Z_{i+1}/Z_i)) \end{array}$$

So, we can simplify the calculation of Z_{i+2} to:

$$Z_{i+2} \ = \ Z_{i+1} \ \cup \ (W_\phi \ \cap \ f(Z_{i+1}/Z_i))$$

Now, when calculating U_{i+2} notice that we only inspect those states in the previous Z_{i+1} which were not added before. It follows, that in the calculation of Z's, no state is ever inspected twice.

3. Keep approximating, until we find a $Z_{k+1} = Z_k$. Then Z_k is our solution.

The improved caluclation of $W_{\mathbf{E}(\phi\ \mathbf{U}\ \psi)}$ is analgous, except we use $g(Z)$ instead of $f(Z)$, where as before $g(Z) = \{s \mid s \in S \text{ and } R(s) \cap Z \neq \emptyset\}$

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.

Also known as labelling

In the literature, the above approach of model checking by calculating W_ϕ is also explained in terms of a *labelling* process. The model checking proceeds by each state s by ϕ if ϕ holds on this state. Note that this is just equivalent to determining the set W_ϕ itself.

4.2 Literature

- 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:

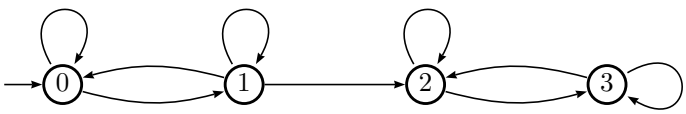
$$p \quad ::= \quad 0 \;\mid\; 1 \;\mid\; \mathit{variable} \mid \neg p \mid p \wedge q \mid p \vee q$$

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 y are two variables) to mean $x \wedge y$. We write $\bar{e}$ to mean the negation of the expression e. So, $\bar{x}\bar{y}$ means $\neg x \wedge \neg y$, whereas $\overline{xy}$ means $\neg(x \wedge y)$.

Derived operators $\Rightarrow$ and $=$ (equality on Boolean values) are defined as usual; furthermore quantifications over Boolean values are defined as follows:

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

Please note that x above is of type Boolean, so we only quantify over 0 and 1.
Consider now this automaton K with just four states:



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
$\overline{x}\overline{y}$	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
$\overline{x}\overline{y}\overline{x'}\overline{y'}$	the loop from 0 to 0
$\overline{x}\overline{y}\bar{x'}y'$	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:

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

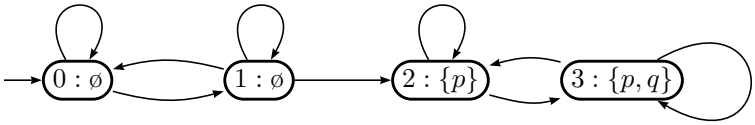
It can be quite short, thus saving space. E.g. all the four arrows between state 2 and 3 can be represented by:

$$xx'$$

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:



We see that p holds on states 2,3, whereas q only holds on 3. We can capture this with a following Boolean function $V(x,y,p,q)$:

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

where $f = g$ above is just boolean equality; so it holds if either fg or $\bar{f}\bar{g}$. The above formula says that p holds only on the states s whose encoding xy has its first component (x) true; whereas q holds on the state whose encoding xy has both its components (x and y) true.

Question: 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[1]

[1] Answer: when you verify a given formula ϕ, you would know what your $Prop$ is. Given this $Prop$, the given concrete state FSA can be reduced to a Kripke structure with labelling L.

5.2 CTL Symbolic Model Checking

Let $K = (S, I, R, Prop, V)$ 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 $s_0 \in W_\phi$ for each initial state $s_0 \in I$.

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 $s \in W_\phi$ if and only if $W_\phi(enc(s)) = 1$, where $enc(s)$ is the vector x, y that represents the state s.

Checking $s_0 \in W_\phi$ is now equivalent to checking if $W_\phi(enc(s_0)) = 1$. If we only have one initial state s_0 then we are done. More generally, if we have multiple initial states, represented by a Boolean formula $init(x, y)$, 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:

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

But this is a set. We need to re-express this set as a Boolean function, which we can do with this one:

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

Whereas the first version of W_p above is a set of states, the second one is indeed a function. It is a Boolean function to be precise; but note that this function equivalently encodes the same set of states.

Suppose we only have one initial state namely s_0 enoced by 00, and 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.

If we have multiple initial states, whose set is symbolically represented by the Boolean function $init(x, y)$, then $K \models p$ can be checked by checking if this:

$$init(x, y) \ \wedge \ W_p(x, y)$$

is *valid*.

Conjunction and Negation

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

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

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

Next Property

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

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

Let us firs re-express this a bit differently (but equivalent):

$$W_{\mathbf{EX}\ \phi} \;=\; \{s \mid (\exists s' :: s' \in R(s) \;\wedge\; s' \in W_\phi)\}$$

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 the validity of this:

$$init(x,y) \;\wedge\; W_{\mathbf{EX}\ \phi}(x,y)$$

Until

Well, assuming you have constructed the Boolean functions $W_\phi(x,y)$ and $W_\psi(x,y)$. Recall that calculate $W_{\mathbf{E}(p\ \mathbf{U}\ q)}$ by iteratively approximating it with Z_i. We will use the second way of iterating these Z_i's, but without the substraction $f(Z_{i+1}/Z_i)$. So:

$$\begin{array}{lcl} Z_0 & = & W_\psi \\ Z_{i+1} & = & Z_i \;\cup\; (W_\phi \cap \{s \mid R(s) \;\cap\; Z_i \neq \emptyset\}) \end{array}$$

The latter can also be written as:

$$Z_{i+1} \;=\; Z_i \;\cup\; (W_\phi \cap \{s \mid (\exists s' :: s' \in R(s) \;\wedge\; s' \in Z_i)\})$$

But we can also represents Z_i with Boolean functions:

$$\begin{array}{lcl} Z_0(x,y) & = & W_\psi(x,y) \\ Z_{i+1}(x,y) & = & Z_i(x,y) \;\vee\; (W_\phi(x,y) \wedge (\exists x', y' :: R(x,y,x',y') \wedge Z_i(x',y'))) \end{array}$$

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.

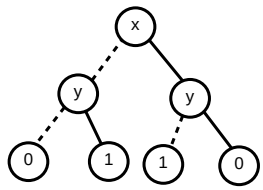
The all-until operator can be handled analogously.

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, we 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.

Here I will explain a complementary approach (to SAT solving). 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 us, due to the exponential size of the table.

A *binary decision tree* is a binary 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.

Let $Node$ be the set of nodes in the tree. We'll use functions $low, high : Node \to Node$. If v is not a leaf, $low(v)$ and $high(v)$ return v's low and respectively high child. The function $ind(v)$ returns the variable (to be precise: 'variable name'q) that decorates the node v, if it is not a leaf. If v is a leaf, $val(v)$ returns its Boolean value, which is either 0 or 1.

To be a binary decision tree, we also need to 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:

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

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:

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

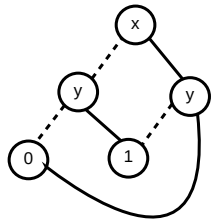
which can be simplified to:

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

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:

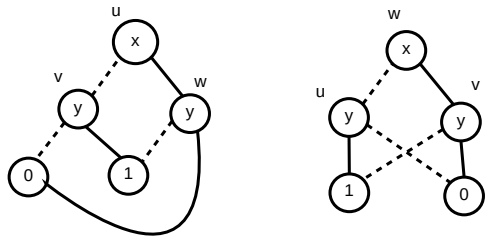
- for any non-leaf node v, $low(v) \neq high(v)$. Otherwise G can be simplified.
- for any distinct nodes v and v', the subgraphs rooted at them are *not* isomorphic. Otherwise G can be simplified. The meaning of 'isomorphic' is explained below.

□

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 the renaming function 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. Actually, just checking whether F has 1 as a leaf will do. But usually you also want to know the values of the variables that would statisfy f. This corresponds to finding a path in F that leads to the leaf 1. This can be checked with e.g. a DFS in $O(N + E)$ time. If you reverse the arrows in F first, which takes $O(N)$, any reverse path from the leaf 1 will lead to the root. Constructing such a reverse path is $O(h)$, where h is the height of the graph F.

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. Hence, v is 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[2].

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:

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

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:

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

[2]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.

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'):

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

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 $low(r_F)$. 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:

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

   where $f = func(F)$ and $g = func(G)$.
   However, because x is the variable at the root, notice that:

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

   And similarly for g. Therefore:

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

   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 Literature

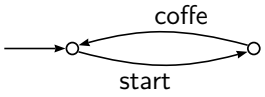
- Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986
- K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell Massachusetts, 1993

Chapter 6

CSP

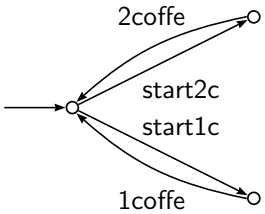
Communicating Sequential Processes (CSP) is a nice formalism you can use to abstractly describe a concurrent system that relies on event synchronization to synchronize its concurrent components. It is introduced by Hoare back in 1978. The original book describing it [5] has been updated (2004) [6] and is available for free at `www.usingcsp.com`.

In CSP a system is described by a 'process'. A CSP process is an *abstraction* of a real program. In this abstraction, the units of execution are *events*. An event has a name, and its execution is atomic (indibisible, cannot be interrupted). For example, a coffee machine can be modelled as a CSP process with two events, start and coffee, which we can describe as the following FSA:



where the event start represents the user action of pushing on the machine's start button, and coffee represents the machine's response of producing coffee.

In CSP there is no concept of program variables as we would have in e.g. C. Events also do not have parameters. You will have to abstract these into undecorated states and unparameterized events such as in the example above. For example, if it is possible for the user of our coffee machine to choose between normal and double quantity coffee, we could model it e.g. like this:



6.1 CSP Syntax and Primitive Operators

However, the above way of describing a process is actually not how CSP does it. It uses a textual expression. The syntax is given below —we will only consider a subset of the full CSP though. A

Process is expressed by one of the following constructs:

STOP	
$Event \rightarrow Process$	– sequential composition
$Process \,\Box\, Process$	– choice
$Process \sqcap Process$	– choice
$Process \,||\, Process$	– parallel composition
$Process/Alphabet$	– hiding
$Name$	
$Name = Process$	– defining a process with an equation

STOP is a process that does not do anything. If P and Q are processes, the composition $a \rightarrow P$ is a process that does the event a and then it behaves as P. The composition $P \Box Q$ is a process that behaves either as P or as Q. CSP has another choice operator, namely $\sqcap$; we will discuss it later. With an equation like:

$$F = a \rightarrow \mathsf{STOP}$$

we can give a name to a process. So, above we define a process named F. The process G belows does b then a then it stops:

$$G = b \rightarrow P$$

We can also define a process recursively, e.g.:

$$Clock = tick \rightarrow (Clock \,\Box\, \mathsf{STOP})$$

defines a process called *Clock* that can do any number of *tick*s. In the above example it is at least intuitively clear what process is meant. The syntax itself does not prevent someone from writing an equation like this:

$$O = O \rightarrow a \rightarrow \mathsf{STOP})$$

or even a bit more subtle:

$$O = (a \,\Box\, O) \rightarrow a \rightarrow \mathsf{STOP})$$

We will avoid this kind of left recursions, as it is not clear which process is meant.

6.1.1 Alphabet, External and Internal Events

An *alphabet* is just a set of events. Some events in a process are *external*. The others are *internal*. External events can be observed and interacted to by the 'environment' of the process, which can be a user other processes. In principle, a process described in CSP only specifies its behavior with respect to its external events (so, its behavior as it would appear to its environment). The *alphabet of a process* P, denoted by αP, is the set of P's external events. It is defined as below:

$\alpha\mathsf{STOP}$	$=$	$\emptyset$
$\alpha(a \rightarrow P)$	$=$	$\{a\} \cup \alpha P$
$\alpha(P \,\Box\, Q)$	$=$	$\alpha P \cup \alpha Q$
$\alpha(P \sqcap Q)$	$=$	$\alpha P \cup \alpha Q$
$\alpha(P \,||\, Q)$	$=$	$\alpha P \cup \alpha Q$
$\alpha(P/A)$	$=$	$\alpha P/A$
$\alpha(P = rhs)$	$=$	$\alpha(rhs[\mathsf{STOP}/P])$ – not left-recursive, assuming no mutual recursion

For a technical reason let's also introduce a new kind of processes: STOP_A is a process that, like STOP, does nothing; but its alphabet is A.

If A is an alphabet, the construct P/A coverts the events in A to internal events. The operator '/' is also called *hiding* or *internalization* operator. For example, if we hide *tick* in the *Clock* process defined above, from the environment's perspective the resulting process would be equivalent to STOP. As another example, consider again F and G defined before:

$G/\{b\}$ is equivalent to F

The alphabet of a process defined by an equation $P = rhs$ can be obtained by calculating the alphabet of the right hand side rhs, where occurrences of P are replaced by STOP. For example, consider again the *Clock* example:

$$\alpha Clock \;=\; \alpha(tick \rightarrow (\mathsf{STOP} \,\Box\, \mathsf{STOP})) \;=\; \{tick\}$$

6.1.2 Parallel Composition

$P||Q$ denotes the *parallel composition* of P and Q. This amounts to interleaved execution of P and Q. However, events that are common in both (so, events in $\alpha P \cap \alpha Q$) must be executed synchronously (together). For example:

$$(a \rightarrow x \rightarrow \mathsf{STOP}) \;||\; (b \rightarrow x \rightarrow \mathsf{STOP})$$

is equivalent to:

$$(a \rightarrow b \rightarrow x \rightarrow \mathsf{STOP}) \,\Box\, (b \rightarrow a \rightarrow x \rightarrow \mathsf{STOP})$$

which is a process that can do either ab or ba (interleaving of respective 'private events') first, then followed by the synchronized execution of the common event x.

In CSP, events can be synchronized by more than just two processes. In the example below, we have three sub-processes that synchronize on the event x:

$$(a \rightarrow x \rightarrow \mathsf{STOP}) \;||\; (b \rightarrow x \rightarrow \mathsf{STOP}) \;||\; (c \rightarrow x \rightarrow \mathsf{STOP})$$

On the other hand we may actually intend x to be only synchronized by the first two sub-processes, and the x in the third process is meant to be a different event which happen to have the same name. We can model it as follows, by using the hiding operator:

$$((a \rightarrow x \rightarrow \mathsf{STOP}) \;||\; (b \rightarrow x \rightarrow \mathsf{STOP}))/\{x\} \;||\; (c \rightarrow x \rightarrow \mathsf{STOP})$$

6.2 Refinement and Specification

Let us write $P \sqsubseteq Q$, pronounced Q *refines* P, to mean that the process Q behaves in such a way that it can be used to replace P. CSP supports such a concept. With it, we can simply use a process to specify how another process should behave. In $P \sqsubseteq Q$, P would take the role of specification and Q the implementation. For example, to a coffee machine can internally be quite complicated. But we can specify it with respect to its external behavior e.g. as follows:

$$\begin{array}{lcl} CoffeeSpec & \sqsubseteq & CoffeeMachine/\{start, coffee\} \\ CoffeeSpec & = & start \rightarrow coffee \rightarrow CoffeeSpec \end{array}$$

But how do we verify it? We will return to this latter. First, let us first try to give a semantical definition to CSP processes so that we can at least have a formal definition of $\sqsubseteq$. I will first show you a so-called *trace semantic*. This semantic is incomplete, but at least intuitive to explain.

6.2.1 Trace Semantic

A *trace* is just a sequence of events. Let us just stick to the notation used in [6] to denote traces:

$\langle\rangle$	is the empty trace
$\langle a\rangle$	is a singleton trace containing a
$\langle a, b, c\rangle$	is a trace with three elements
$s \hat{} t$	concatenation of two traces
$s{\restriction}A$	projection or restriction: the trace obtained by throwing away events *not* in A

Furthermore, if A is an alphabet, then A^* denotes all finite traces over the events in A.

Let's use traces to model executions of a process. Each is a sequence of events that the process can produce. For example the process:

$$R_1 \;=\; a \rightarrow b \rightarrow \mathsf{STOP}$$

when executed can exibit three traces: $\langle\rangle$, $\langle a\rangle$, and $\langle a, b\rangle$. The set of traces that a process P can exibit is denoted by $\mathbf{traces}(P)$. Another example:

$$R_2 \;=\; (a \rightarrow a \rightarrow \mathsf{STOP}) \,\Box\, (b \rightarrow a \rightarrow \mathsf{STOP})$$

$\mathbf{traces}(R_2) = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, a\rangle, \langle b, a\rangle, \}$ In general, the traces of a process can be calculated as follows:

$$\begin{array}{lcl}
\mathbf{traces}(\mathsf{STOP}) & = & \emptyset \\
\mathbf{traces}(a \rightarrow P) & = & \{\langle\rangle\} \;\cup\; \{\langle a\rangle \hat{} s \mid s \in \mathbf{traces}(P)\} \\
\mathbf{traces}(P \,\Box\, Q) & = & \mathbf{traces}(P) \;\cup\; \mathbf{traces}(Q) \\
\mathbf{traces}(P \,\sqcap\, Q) & = & \mathbf{traces}(P) \;\cup\; \mathbf{traces}(Q) \\
\mathbf{traces}(P/A) & = & \{s{\restriction}(\alpha P/A) \mid s \in \mathbf{traces}(P)\}
\end{array}$$

Notice in particular how the semantic of $a \rightarrow P$ is defined. Its set of traces consists not only of what it can 'maximall' do, but also all the prefixes in between. For example, the traces of $a \rightarrow b \rightarrow \mathsf{STOP}$ do not only consist of the maximal $\langle a, b\rangle$, but also all its prefixes $\langle\rangle$ and $\langle a\rangle$. In fact, it is a general property of our trace semantic:

Theorem 6.2.1 : Prefix Closed-ness
If s is a prefix of t and $t \in \mathbf{traces}(P)$, then $s \in \mathbf{traces}(P)$.
□

What is the traces of $P||Q$? Note that that s is a trace of this composition if and only if when projected on αP it must be a trace of P, and similarly when projected on αQ it is a trace of Q. This leads to this formula:

$$\mathbf{traces}(P||Q) \;=\; \{s \mid s \in (\alpha(P||Q))^*,\; s{\restriction}\alpha P \in \mathbf{traces}(P),\; s{\restriction}\alpha Q \in \mathbf{traces}(Q)\;\}$$

For a process defined by a non-recursive equation $P = rhs$, its traces is simply the traces of the right hand side formula rhs. If the equation is recursive, we define its set of traces as the smallest solution to:

$$\mathbf{traces}(P) \;=\; \mathbf{traces}(rhs)$$

For example, consider this variation of $Clock$:

$$Clock_2 \;=\; tick \rightarrow tock \rightarrow (Clock_2 \,\Box\, \mathsf{STOP})$$

Its traces is then the smallest solution of:

$$\mathbf{traces}(Clock_2) \;=\; \mathbf{traces}(tick \rightarrow tock \rightarrow (Clock_2 \,\Box\, \mathsf{STOP}))$$

Using the other definitions, we can work out the right hand side, and reduce the above equation to the following:

$$\mathbf{traces}(Clock_2) \;=\; \{\langle\rangle, \langle a\rangle\} \;\cup\; \{\langle a, b\rangle \,\hat{}\, s \mid s \in \mathbf{traces}(Clock_2)\;\}$$

The smallest solution of the above equation is (which you probably already guess):

$$\mathbf{traces}(Clock_2) \;=\; \{\langle a, b\rangle^k \mid k \geq 0\}$$

Now we can define what $\sqsubseteq$ means. Let us choose for a rather conservative definition. Q refines P if it *cannot* do something that P won't be able to do. So, if P cannot blow up your computer, and $P \sqsubseteq Q$, it follows that Q won't blow up the computer either. Note that by this definition, a process that does nothing will refine any specification. We will deal with this later. For now, we define:

Definition 6.2.2 : Trace-based Refinement

$$P \sqsubseteq Q \;=\; \alpha P{=}\alpha Q \;\wedge\; \mathbf{traces}(Q) \subseteq \mathbf{traces}(P)$$

□

Unfortunately, as we have seen in the above example, the set of traces of even a small process can be infinitely large. So, verifying $P \sqsubseteq Q$ by directly using the above definition is not going to work.

6.3 FSA Semantic

An FSA can also be used to describe an event-based system, e.g. as the coffee machine examples at the beginning of this Chapter. You can perhaps even say that it is a more direct way of describing such a process, since the states and how the events move the process from one state to another are described explicitly. In any case, FSAs seem to be expressive enough to be used as the semantic of CSP processes.

Let's take FSAs with structures as described in Definition 3.1.1. In particular, the arrows are labelled. Such an FSA M is described by a tuple (S, s_0, A, R) where S is a set of states, s_0 is an initial state (we will only use one initial state), A is the alphabet of M (its set of events), and R is a function that described the arrows in M.

Additionally, we introduce a special event denoted by τ representing internal event (there may be different sorts of internal events, but we will not distinguish between them). An arrow in M can also be labelled with τ. So, if there is an arrow from the state s to t labelled with τ then $t \in R(s, \tau)$.

A trace of M is a sequence of events that can be produced by following a path in M, starting from its initial state. We however remove τ-events from such a trace (in other words, a 'trace' of M consists only of events in A, and does not contain any τ). We write $\mathbf{trace}(M)$ to denote the set of all traces that M can produce.

Let $\mathbf{fsa}(P)$ denote the FSA that corresponds to P. We want this FSA to equivalently describe P, in the sense that:

$$\mathbf{trace}(\mathbf{fsa}(P)) \;=\; \mathbf{trace}(P) \tag{6.1}$$

This FSA of P can be constructed as follows:

1. $\mathbf{fsa}(\mathsf{STOP}) \;=\; (\{s_0\}, s_0, \emptyset, R)$; the R part is irrelevant because the alphabet is empty, so it cannot specify any arrow.

2. Let $\mathbf{fsa}(P) = (S, s_0, A, R)$. Then $\mathbf{fsa}(a \rightarrow P)$ is constructed as follows:

   $$(\{t_0\} \cup S, t_0, \{a\} \cup A, R')$$

   where R' is R, but to represent the sequencing, it is extended with the mapping $R(t_0, a) = \{s_0\}$. Notice that the new FSA has a new initial state as well.

3. Let $\mathbf{fsa}(P) = (S_1, s_1, A_1, R_1)$ and $\mathbf{fsa}(Q) = (S_2, s_2, A_2, R_2)$. Let $((S_3, s_3, A_3, R_3)$ be the FSA of $P \,\Box\, Q$. Let us first construct it as below. The construction is intuitive, but is actually not entirely correct. We will return to this later.

   (a) $A_3 = A_1 \cup A_2$

   (b) $S_3 = \{s_3\} \cup S_1 \cup S_2$. Notice that s_3 is not the new initial state.

   (c) R_3 is the 'union' of R_1 and R_2. Additionally, to represent the choices we additionally we add τ-arrows from the new initial state s_3 to s_1 and s_2. So:

$$R_3(t,a) = \begin{cases} R_1(t,a) & \text{, if } t \in S_1 \\ R_2(t,a) & \text{, if } t \in S_2 \end{cases}$$

$$R_3(s_3,\tau) = \{s_1, s_2\}$$

4. The FSA of $P \sqcap Q$ is constructed in the same way as $P \,\Box\, Q$.

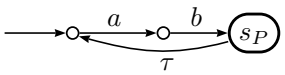
5. Let $\mathbf{fsa}(P) = (S, s_0, A, R)$. Then $\mathbf{fsa}(P/B)$ is $(S, s_0, A/B, R')$, where R' is obtained from R by replacing arrows labelled with $a \in B$ with arrows labelled with τ:

$$\begin{aligned} R'(s,a) &= R(s,a) \quad \text{if } a \notin B \\ R'(s,\tau) &= R(s,a) \quad \text{if } a \in B \end{aligned}$$

6. The FSA of $P||Q$ is constructed by taking the interleaving product of $\mathbf{fsa}(P)$ and $\mathbf{fsa}(Q)$. This is as discussed in Subsection 3.2.2 (more specifically, the variant with synchronized events).

7. When P is defined by an equation $P = rhs$, its FSA is just the FSA of rhs. However, if the equation is recursive (but not left recursive), in the construction of $\mathbf{fsa}(rhs)$ we replace every occurrence of P in rhs by a an automaton that only consist of a single state s_P, with no arrow. Then, we add a τ arrow from this s_P to the initial state of $\mathbf{fsa}(rhs)$.

   For example, the FSA of $P = a \to b \to P$ is (shown graphically):



As a side note, if we allow quantified equations in CSP, we can write something like:

$$\begin{aligned} Aircraft_0 &= (flyUp \to Aircraft_1) \,\Box\, (landing \to \mathsf{STOP}) \\ Aircraft_{i+1} &= (flyUp \to Aircraft_{i+2}) \,\Box\, (flyDown \to Aircraft_i) \end{aligned}$$

Which gives us additional expressiveness; but, such processes cannot be expressed with a *finite* state automaton.

6.3.1 Refinement Checking with FSA

Since $\mathbf{FSA}(P)$ has the same traces as P, the problem of checking $P \sqsubseteq Q$ can be reduced to checking traces inclusion in the corresponding FSAs:

Theorem 6.3.1 :

$$P \sqsubseteq Q = \mathbf{traces}(\mathbf{fsa}(Q)) \subseteq \mathbf{traces}(\mathbf{fsa}(P))$$

□

But note this was using $\sqsubseteq$ defined in terms of traces (Definition 6.2.2).

Let us first define some concepts and notations. Let M be an FSA. We write $\mathbf{initials}(M)$ is the set of events that M can do as its first event. If $M = (S, s_0, A, R)$ is *deterministic* FSA , its initials can be calculated as follows:

$$\mathbf{initials}(M) \;=\; \{a \mid a \in A, R(s_0, a) \neq \emptyset\} \tag{6.2}$$

Note by the way that every non-deterministic FSA can be systematically converted into an equivalent deterministic FSA.

Analogously, if s is a state, $\mathbf{initials}_M(s)$ is the set of first events that M can do when it is it the state s:

$$\mathbf{initials}_M(s) \;=\; \{a \mid a \in A, R(s, a) \neq \emptyset\} \tag{6.3}$$

If σ is a trace of M, at the end of σ M must be in a certain state t. Since M was assumed to be deterministic, there is only one possible such t for each σ, denoted by $\mathbf{endstate}_M\sigma$. The initials of σ is defined as the initials of t:

$$\mathbf{initials}_M(\sigma) \;=\; \mathbf{initials}_M(\mathbf{endstate}_M(\sigma)) \quad \text{, assuming } \sigma \in \mathbf{traces}(M) \tag{6.4}$$

Now, the theorem below can be proven, which shows an alternative way to check traces inclusion (crucial for its proof is the observation that traces of an FSA is prefix-closed):

Theorem 6.3.2 :
Let M and L be two FSAs with the same set of events.

$$\mathbf{traces}(M) \subseteq \mathbf{traces}(L) \;=\; (\forall \sigma : \sigma \in \mathbf{traces}(M \cap L) : \mathbf{initials}_M(\sigma) \subseteq \mathbf{initials}_L(\sigma))$$

where the intersection $M \cap L$ is as defined in Definition 3.2.1.
□

The above still does not help us, because it requires us to quantify over $\mathbf{traces}(M \cap L)$, which can be infinite. However, if we first reduce M and K so that they are now deterministic, then as pointed out before, executing σ would bring M to a unique state s and L to unique state t. The initials of σ of each FSA is just the same as the initials of this s respectively t. Furthermore, note that in the construction of $M \cap L$ as in Definition 3.2.1 is state of $M \cap L$ is actually a pair (v, u) where v is a state of M and u is a state of L. We can prove the following:

Theorem 6.3.3 :
Let M and L be two deterministic FSAs with the same set of events.

$$\mathbf{traces}(M) \subseteq \mathbf{traces}(L) \;=\; (\forall (v, u) : (v, u) \in \mathbf{states}(M \cap L) : \mathbf{initials}_M(v) \subseteq \mathbf{initials}_L(u))$$

where the intersection $\mathbf{states}(M \cap L)$ is the set of all states of $M \cap L$.
□

Importantly, notice that now we quantify over the states of $M \cap L$. There are only finitely many of them. Consequently, the above checking can be done in finite time. Consequently, refinement checking based on the trace semantic can also be done in finite time. Figure 6.1 shows a DFS-based algorithm to do this; Figure 6.1 shows an iterative version of it.

```
--  M = (S_2, s_2, A, R_2) and M = (S_1, s_1, A, R_1) are deterministic FSAs
checkInclusion(M, L) {
   checked := emptyset ;
   DFS(s_2, s_1)
   where
   DFS(t_2, t_1) {
      if((t_2, t_1) ∈ checked) return ;
      checked := checked ∪ {(t_2, t_1)} ;
      if(initials_M(s_2) ⊈ initials_L(s_1)) throw Violation ;
      forall (u_2, u_1) such that (∃a : a∈A : u_1∈R_1(t_1, a) ∧ u_2∈R_2(t_2, a))
         DFS(u_2, u_1)
   }
}
```

Figure 6.1: A DFS-based recursive algorithm to check traces-inclusion.

```
--  M = (S_2, s_2, A, R_2) and M = (S_1, s_1, A, R_1) are deterministic FSAs
checkInclusion(M, L) {
   checked := ∅ ;
   pending := {(s_2, s_1)}  ;
   while(pending ≠ ∅) {
      (t_2, t_1) := an element retrieved from pending ;
      checked := checked ∪ {(t_2, t_1)} ;
      if(initials_M(s_2) ⊈ initials_L(s_1)) throw Violation ;
      pending := pending
                 ∪
                 {(u_2, u_1) | (∃a : a∈A : u_1∈R_1(t_1, a) ∧ u_2∈R_2(t_2, a) ∧ (u_2, u_1)∉checked)}
   }
}
```

Figure 6.2: An iterative algorithm to check traces-inclusion.

6.4 Enforcing Progress

We have chosen for a conservative refinement relation. It makes sure that e.g. an implementation Q won't blow up anything unless it is allowed by its specification P. Unfortunately, it does not enforce progress. For example a Q that does nothing will by our definition satisfy any specification.

Enforcing progress does bring some new complication. Consider the parallel composition:

$$(a \rightarrow b \rightarrow \mathsf{STOP}) \;||\; (b \rightarrow a \rightarrow \mathsf{STOP})$$

Obviously this composition will get stuck. Both a and b are common events of the two subprocesses, so they have to synchronize on them. But this is not possible. In other words, the composition comes to a deadlock after $\langle\rangle$. Consider now this composition:

$$(a \rightarrow b \rightarrow \mathsf{STOP}) \;||\; ((a \rightarrow \mathsf{STOP}) \;\Box\; (b \rightarrow \mathsf{STOP}))$$

Obviously, it will deadlock after the first event. But will is also deadlock at $\langle\rangle$? That depends on how the choice in $(a \;\Box\; b)$ is made. If the subprocess $(a \;\Box\; b)$ itself internally decides which choice it takes, then yes: the above composition *may* deadlock at $\langle\rangle$.

But if it is the 'environment' that dictates the choice, the the above composition won't deadlock at $\langle\rangle$. So, when progress is an issue, then we need to distinguish between 'external' and 'internal' choices. In CSP, the $\Box$ operator expresses external choice, whereas $\sqcap$ expresses internal choice. So, the above composition indeed won't deadlock at $\langle\rangle$, whereas its variant below may deadlock at $\langle\rangle$:

$$(a \rightarrow b \rightarrow \mathsf{STOP}) \;||\; ((a \rightarrow \mathsf{STOP}) \;\sqcap\; (b \rightarrow \mathsf{STOP}))$$

When making its choice, CSP assumes the environment to only look at what it can do next (we do not assume the evironment to be smart enough to look beyond that). Consider this example, where $\alpha P = \alpha Q = \{a, b, c\}$:

$$P \;||\; Q \quad \text{where: } Q \;=\; ((a \rightarrow b \rightarrow \mathsf{STOP}) \;\Box\; (a \rightarrow c \rightarrow \mathsf{STOP}))$$

Q offers a choice between doing $\langle a, b\rangle$ or $\langle a, c\rangle$. But P cannot see that far ahead. It only sees that the first possible event to do is a. That means that Q itself will then have to make the choice whether it then go to the first branch or the other. In other words, this choice becomes internal. It could be that Q takes the first branch, and thus its next possible event is b. If $P = a \rightarrow c \rightarrow \mathsf{STOP}$ it will then deadlock at that point. With respect to P, Q behaves actually as:

$$Q \;=\; a \rightarrow ((b \rightarrow \mathsf{STOP}) \;\sqcap\; (c \rightarrow \mathsf{STOP}))$$

The simplistic trace semantic given in Subsection 6.2.1 cannot distinguish between external and internal choice. In otherwords, the traces of $P \;\Box\; Q$ and $P \sqcap Q$ are the same. Below we will show how to extend this semantic so that we can also enforce progress.

6.4.1 Refusals

Consider these processes:

$$\begin{array}{lcl} P_1 & = & (a \rightarrow \mathsf{STOP}) \;\Box\; (b \rightarrow \mathsf{STOP}) \\ P_2 & = & a \rightarrow b \rightarrow \mathsf{STOP} \\ P_3 & = & (a \rightarrow \mathsf{STOP}) \;\sqcap\; (b \rightarrow \mathsf{STOP}) \end{array}$$

Suppose we compose them in parallel with an environment whose alphabet is also $\{a, b\}$. The process P_1 will not refuse (unable to synchronize) any first event from the environment, whereas P_2 will refuse b. P_3 may refuse a, if it is the only event the environment can do, and analogously b. However, P_3 *cannot* refuse *both* a and b. That is, if the environment is able to either synchronize

with a or b, then despite the internal choice made by P_3, the composition $P||env$ will be able to do its first event.

A *refusal* of P is a subset A of its alphabet (thus, it ia a set of events) such that when the environment of P offers to synchronize over any event in A (P may choose) as the first event to be executed together with P , it is still possible for P, due to some internal choices it makes, to come to some internal state where P cannot do any event in A (it deadlocks on A).

The refusals-set of P, denoted by $\mathbf{refusals}(P)$ is the set of *all* P's refusals. Consider again the above examples:

$$\begin{array}{lcl} \mathbf{refusals}(P_1) & = & \{\emptyset\} \\ \mathbf{refusals}(P_2) & = & \{\emptyset, \{b\}\} \\ \mathbf{refusals}(P_3) & = & \{\emptyset, \{a\}, \{b\}\} \end{array}$$

Notice that a refusal is a set, and **refusals** is a set of sets.

One more example:

$$P_4 \;=\; ((a \rightarrow \mathsf{STOP}) \,\Box\, (b \rightarrow \mathsf{STOP})) \;\sqcap\; ((c \rightarrow \mathsf{STOP}) \,\Box\, (d \rightarrow \mathsf{STOP}))$$

If the environment offers $\{a, b, c, d\}$, obviously P_4 won't deadlock. But if the environment only offers $\{a, b\}$ then P_4 may deadlock, namely if it chooses to do $c \,\Box\, d$. Similarly, $\{c, d\}$ can also be refused. However, notice that $\{a, c\}$, or $\{a, b, c\}$, or $\{b, c\}$ will not be refused. P_4's full set of refusals consists of:

1. $\{a, b\}$ and all its subsets.
2. $\{c, d\}$ and all its subsets.

We can generalize this concept. A set $A \subseteq \alpha P$ is a refusal of a trace s, if it is still possible for P that after it does s, due to some internal choices it makes along the way or after s, to come to some internal state where P cannot do any event in A (it deadlocks on A).

Similarly, we define $\mathbf{refusals}(P/s)$ as the set of *all* s's refusals in P. And of course, $\mathbf{refusals}(P) = \mathbf{refusals}(P/\langle\rangle)$. Refusals have the following properties:

1. Trivially, empty set is always a refusal. For any trace s of P:

$$\emptyset \in \mathbf{refusals}(P/s) \tag{6.5}$$

2. If P can refuse A, it can refuse any subset of A:

$$B \in \mathbf{refusals}(P/s) \;\wedge\; A \subseteq B \;\Rightarrow\; A \in \mathbf{refusals}(P/s) \tag{6.6}$$

3. If after doing s, P cannot refuse $a \in \alpha P$, then sa must be possible:

$$(\forall A :: a \notin \mathbf{refusals}(P/s)) \;\Rightarrow\; s \,\hat{}\, \langle a \rangle \in \mathbf{traces}(P) \tag{6.7}$$

In particular the last property gives us a hint on how to enforce progress: the left side of $\Rightarrow$ can be seen as a requirement for P to progress from s to sa. Or generally, it appears that specifying refusals can be used as a way to express progress requirement. So, let's do that: we will extend our CSP semantic to also include the refusals. Somewhat misleadingly, the new semantic is called the 'failures' of P.

Definition 6.4.1 : Failures

$$\mathbf{failures}(P) \;=\; \{(s, X) \mid s \in \mathbf{traces}(P),\; X \in \mathbf{refusals}(P/s)\;\}$$

□

Notice that since $\emptyset$ is always a refusal, all traces of P are automatically included in its failures.

Like **traces**, **failures** is also prefix-closed. But furthermore, with respect to the refusals it is also subset-closed:

Theorem 6.4.2 : Downward Closed-ness
If (t, B) is a failure of P (it is a member of $\mathbf{failure}(P)$), then for any prefix s of t and any $A \subseteq B$, (s, A) is also a failure of P.
□

And now we can define a stronger concept of refinement, namely one based on failures:

Definition 6.4.3 : Failures-based Refinement

$$P \sqsubseteq Q \;=\; (\alpha P = \alpha Q) \;\wedge\; \mathbf{failures}(Q) \subseteq \mathbf{failures}(P)$$

□

Under this concept of refinement, you can no longer trivially implement e.g. $a \rightarrow \mathsf{STOP}$ with $\mathsf{STOP}_{\{a\}}$. Another example: you can refine e.g.:

$$(a \rightarrow \mathsf{STOP}) \sqcap (b \rightarrow \mathsf{STOP})$$

by reducing its non-determinism, to:

$$(a \rightarrow \mathsf{STOP}) \,\Box\, (b \rightarrow \mathsf{STOP})$$

This is as expected. This refinement is also possible with the old trace semantic. But with the failure semantic, the other way around is no longer true.

6.5 Refinement Checking Revisited

To check refinement under the new semantic we will have to revisit our refinement checking approach (Section 6.3). To still do the checking through FSAs we have to make sure that our FSA semantic (Section 6.3) translates every CSP process P to an 'equivalent' FSA, which now mean that $\mathbf{fsa}(P)$ should define the same set of failures as P itself. But before we can come to that, a whole bunch of supporting concepts need to be introduced or redefined first.

6.5.1 Fixing the FSA-semantic

The definition of $\mathbf{fsa}(P)$ in Section 6.3 is good, except for the $\Box$ and $\sqcap$ constructs. They are not distinguished. For the new semantic they should be distinguished. We will keep the definition of $\mathbf{fsa}(P \sqcap Q)$ as it was, namely the same as the *old* definition of $\mathbf{fsa}(P \,\Box\, Q)$.

We redefine $\mathbf{fsa}(P \,\Box\, Q)$ as follows:

Let $\mathbf{fsa}(P) = M_1 = (S_1, s_1, A_1, R_1)$ and $\mathbf{fsa}(Q) = M_2 = (S_2, s_2, A_2, R_2)$. Let $((S_3, s_1, A_3, R_3)$ be the FSA of $P \,\Box\, Q$; it is constructed as follows:

1. $S_3 = S_1 \cup S_2$
2. R_3 is just the 'union' of R_1 and R_2. However, we make s_1 the new initial state. We keep all outgoing arrows from s_1 as in M_1; but additionally, we add to it M_2's initial arrows. So:

$$\begin{array}{lcll} R_3(t,a) & = & R_1(t,a) & \text{, if } t \in S_1/\{s_1\} \\ R_3(u,b) & = & R_2(u,b) & \text{, if } u \in S_2/\{s_2\} \\ R_3(s_1,a) & = & R_1(s_1,a) \cup R_2(s_2,a) & \end{array}$$

6.5.2 Fixing the definition of initials

Let $M = (S, s_0, A, R)$ be an FSA. The initials set of s ($\mathbf{initials}_M(s)$) is defined as the set of all events that M can do next after doing s. However, recall that the definitions in (6.3) and (6.2) assume M to be deterministic. For refinement checking with the trace semantic, this is good enough. With the failure semantic, non-determinism matters. So we have to redefine it.

We write $s \xrightarrow{\sigma} t$ to mean that from the state s M can reach the state t by doing the trace σ —doing τ-arrows in between is allowed. We write $s \xrightarrow{\tau^+} t$ to mean that the state t can be reached from the state s by following one or more τ-arrows. We call such a t a τ-closure of s. We define:

$$\mathbf{xchoices}_M(s) \;=\; \{a \mid a \in A,\; R(s,a) \neq \emptyset\} \tag{6.8}$$

It is the set of all events (excluding τ-event!) labelling the outgoing arrows from s (this is what **initials** was in the previous definition). The initials of s are the events that M can do either immediately when it is at the state s, or after doing some τ-steps:

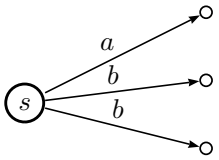
$$\mathbf{initials}_M(s) \;=\; \mathbf{xchoices}_M(s) \;\cup\; (\cup t : s \xrightarrow{\tau^+} t : \mathbf{xchoices}_M(t)) \tag{6.9}$$

Note that this set can be calculated.

6.5.3 Refusals of a state in FSA

A refusal of a state s is a set of events that M can deadlock when it is in that state. The refusals set can be calculated as below. We will distinguish three cases of s. The first two are actually special cases of the third. I show them to just to help you towards understanding the third one.

1. Suppose the state s that does not have any outgoing τ-arrow (except to itself), e.g. as the situation below:

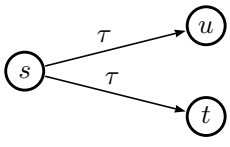


   The choice of which arrows to follow will be considered as made by the environment. So, it represents external choices. However, the environment can only choose based on the event. So, the environment can choose between doing a or b; but it cannot choose which of the two b-arrows to follow. The latter is decided internally by M.

   In the above example, whan on s, M will thus not refuse any non-empty subset of $\{a, b\}$. Any subset X that does not intersect with $\{a, b\}$ is refused. So, in this case:

$$\mathbf{refusals}_M(s) \;=\; \{X \mid X \subseteq A,\; X \cap \mathbf{xchoices}_M(s) = \emptyset\;\}$$

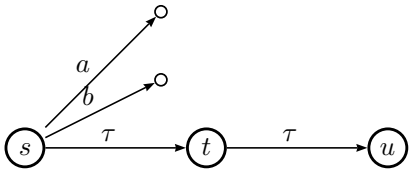
2. Suppose s only have outgoing τ-arrows, e.g. as below:



   In this case, when at s, M can refuse any X_1 refused by u and also any X_2 refused by t. But note that $X_1 \cup X_2$ cannot be refused. So:

$$\mathbf{refusals}_M(s) \;=\; \mathbf{refusals}_M(t) \;\cup\; \mathbf{refusals}_M(u)$$

3. Finally, as the third case consider when s has both τ and non-τ arrows. E.g. as below:



As in the second case, at s the FSA M can refuse any X_1 that can be refused by t, and any X_2 that can be refused by u. But it can also refuse any X that does not include a or b, namely if M insists on not taking the τ arrows. The refusals of s is:

$$\mathbf{refusals}_M(s) \;=\; \{X \mid X \subseteq A,\; X \cap \mathbf{xchoices}_M(s) = \emptyset\;\} \;\cup\; \mathbf{refusals}_M(t) \;\cup\; \mathbf{refusals}_M(u)$$

Finally, we can now define what the failures of an FSA:

Definition 6.5.1 : FAILURES OF FSA
Let $M = (S, s_0, A, R)$ be an FSA.

$$\mathbf{failures}(M) \;=\; \{(\sigma, X) \mid \sigma \in \mathbf{traces}(M), s_0 \xrightarrow{\sigma} t, X \in \mathbf{refusals}_M(t)\;\}$$

We claim that our **fsa** semantic preserves the failures:

$$\mathbf{failures}(P) \;=\; \mathbf{failures}(\mathbf{fsa}(P)) \tag{6.10}$$

Unfortunately, we cannot prove this because we have not given a formal definition of $\mathbf{failures}(P)$ (the way we did for $\mathbf{traces}(P)$. Alternatively, if you consider the way $\mathbf{fsa}(P)$ and $\mathbf{failures}(M)$ have been defined as reasonable, we can just as well take the above equation as the definition of $\mathbf{failures}(P)$.

So now refinement (Definition 6.4.3) can be re-expressed as follows:

$$P \sqsubseteq Q \;=\; \mathbf{failures}(\mathbf{fsa}(Q)) \subseteq \mathbf{failures}(\mathbf{fsa}(P)) \tag{6.11}$$

6.5.4 Fixing DFSA reduction

So, as shown above, refinement checking boils down to checking if the failures of one FSA are included in that of another. To do the latter, we prefer to have deterministic FSAs.

As noted before, any non-deterministic FSA M can be converted to an 'equivalent' deterministic FSA M'. But the standard way of doing such reduction only guarantees that M and M' are traces-equivalent (they produce the same traces). But now we need to make sure that they are also failures equivalent.

Let $M = (S, s_0, A, R)$ be an FSA, let $\mathbf{dt}(M)$ be a function that construct a deterministic version of M. The construction is shown below; it is the usual way to convert an NDFSA to a DFSA, but it is *extended*. The resulting deterministic FSA is extended with labelling. It has the following structure: $M' = (S', t_0, A, R', lab)$. The elements of M' are as usual, except for the new element $lab : S' \to pow(pow(A))$ that labels every state of M' with a set of refusals. The construction of M' is shown below; each state in S' will be a subset of S.

1. The initial state t_0 is $\{s_0\} \cup \{t \mid s_0 \xrightarrow{\tau^+} t \text{ in } M\;\}$.

2. We label t_0 with the union of all refusals-set of its members. So:

$$lab(t_0) \;=\; (\cup t : t \in t_0 : \mathbf{refusals}_M(t))$$

3. We start with $S' = \{t_0\}$ and R' such that we have no arrows. So:

   $$R'(t_0, a) = \emptyset \quad \text{, for all } a$$

4. For each state $T \in S'$, and for each $a \in A$ such that there is at least one arrow in M from some state in T labelled with a, we construct a new state, namely:

   $$U \;=\; (\cup t : t \in T : R(t, a))$$

   U is then added to S', if it is not already there.

   We add an arrow from T to U, labelled with a. So, we add U to $R(T, a)$.

   The label of U is:

   $$lab(U) \;=\; (\cup u : u \in U : \mathbf{refusals}_M(u))$$

   We repeat this step until no new states can be added to S'; then we are done.

For a deterministic M', recall that a trace σ will lead M' to a unique state u (σ fully determines the state of M' at the end of σ). This u is denoted by $\mathbf{endstate}_{M'}(\sigma)$. Recall that for such M', and a trace σ of M', $\mathbf{initials}(\sigma)$ is defined to be the initials of its end-state. Analogously, we define:

$$\mathbf{refusals}_{M'}(\sigma) \;=\; lab(\mathbf{endstate}_{M'}(\sigma)) \tag{6.12}$$

We claim that the failures of M can be re-expressed in terms of its deterministic version M':

$$\mathbf{failures}(M) \;=\; \{(\sigma, X) \mid \sigma \in \mathbf{traces}(\mathbf{dt}(M)),\; X \in \mathbf{refusals}_{\mathbf{dt}(M)}(\sigma)\;\} \tag{6.13}$$

For checking failures inclusion we can prove the following theorem, which is the analogous of Theorem 6.3.2 —the proof relies on the prefix-closed-ness of traces and downward-closed-ness of refusals.

Theorem 6.5.2 :
Let M and L be two FSAs with the same set of events, $M' = \mathbf{dt}(M)$, and $L' = \mathbf{dt}(L)$.

$$\begin{array}{l}\mathbf{failures}(M) \subseteq \mathbf{failures}(L)\\ =\\ (\forall \sigma : \sigma \in \mathbf{traces}(M' \cap L') : \mathbf{initials}_{M'}(\sigma) \subseteq \mathbf{initials}_{L'}(\sigma) \;\wedge\; \mathbf{refusals}_{M'}(\sigma) \subseteq \mathbf{refusals}_{L'}(\sigma))\end{array}$$

□

Again, such a theorem cannot directly help us: to check the inclusion we have to quantify over the traces of $M' \cap L'$, whose number can be infinite.

And finally, the analogous of Theorem 6.3.3:

Theorem 6.5.3 :
Let M and L be two FSAs with the same set of events, $M' = \mathbf{dt}(M)$, and $L' = \mathbf{dt}(L)$. Let $lab_{M'}$ and $lab_{L'}$ be the labelling functions of M' respectively L'.

$$\begin{array}{ll}\mathbf{failures}(M) \subseteq \mathbf{failures}(L) & \\ = & \\ (\forall (V, U) : (V, U) \in \mathbf{states}(M' \cap L') : & \mathbf{initials}_{M'}(V) \subseteq \mathbf{initials}_{L'}(U)\\ & \wedge\\ & lab_{M'}(V) \subseteq lab_{L'}(U)\;)\end{array}$$

□

The above can be checked systematically, namely by the same algorithm as shown in Figures 6.1 and 6.2, except that we need to extend the check for violation. By the Theorem above, we now need to check:

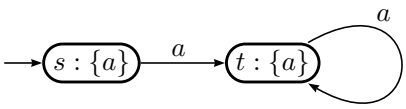
$$\mathbf{initials}_{M'}(V) \subseteq \mathbf{initials}_{L'}(U) \quad \wedge \quad lab_{M'}(V) \subseteq lab_{L'}(U)$$

which indeed can be computed straightforwardly. Thus, we have effectively shown how refinement under the failures semantic can be done.

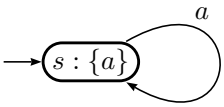
6.5.5 Some notes

The refinement checking algorithms presented here are simpler variations of the one given by Roscoe [12]. In particular, we do not consider the issue of divergence here.

The coversion to deterministic FSA as explained in Subsection 6.5.4 corresponds to what in [12] called pre-normal-form transformation. Consider this FSA, which is deterministic:



The alphabet is $\{a\}$. It has two states, s and t, decorated (to save space) only their respective maximal refusals. So, at each at this state, the FSA can either continue doing an a, or to simply stop (because it can refuse $\{a\}$). Our **dt** reduction will not reduce such an FSA further, yet Roscoe pointed out that the two states are in a way equivalent: all their future traces and refusals are indistinguishable. So, he defines the normal form to be the minimum DFSA such that such equivalent states are merged into one. For the above example, the normal form is:



Notice that indeed both FSAs have the same failures sets.

Roscoe orginal algorithm requires that when checking $P \sqsubseteq Q$ that the FSA of P (the specification-side of the refinement) has been normalized as shown in the example above. However, for failures-based checking I think that pre-normal forms will do. Note that using our checking algorithm we can show that each of the above two automata above is refined by the other.

Another difference is that Roscoe does not require the FSA of Q (the implementation-side of the refinement) to be reduced to a DFSA. In contrats, we require both P and Q to be reduced to DFSAs. We did this in favor of simplicity (so it is easier to explain the algorithm). It is possible to drop this requirement for Q, but you also need to tweak the algorithms. You are advised to look at [12] –it should give you enough clue on what to change.

6.6 Literature

- C.A.R. Hoare. *Communicating Sequential Processes*. 2004. Online at `www.usingcsp.com`
- A. W. Roscoe. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378. Prentice Hall, 1994. available online at `www.cs.ox.ac.uk/oucl/work/bill.roscoe/publications/50.ps`

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.W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[5] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[6] C.A.R. Hoare. *Communicating Sequential Processes*. 2004. Online at `www.usingcsp.com`.

[7] Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.

[8] 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.

[9] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking java programs via guarded commands. Technical Report SRC Technical Note 1999-002, Compaq, 1999. available online at `http://research.microsoft.com/en-us/um/people/leino/papers/SRC-1999-002.pdf`.

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

[11] I.S.W.B. Prasetya. *Introduction to Progamming Logic*. Dept. of Inf. & Comp. Sciences, Utrecht Univ., 2012. Lecture Notes of the course Software Testing and Verification, online available at `www.cs.uu.nl/docs/vakken/pc`.

[12] A. W. Roscoe. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378. Prentice Hall, 1994. available online at `www.cs.ox.ac.uk/oucl/work/bill.roscoe/publications/50.ps`.

[13] 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.