University of Sheffield
Department of Computer Science

The Theory and Practice of Specification Based Software Testing

Gilbert Thomas Laycock

Submitted towards the degree of
Doctor of Philosophy
September 1992, April 1993

The Theory and Practice of Specification Based Software Testing

Gilbert Thomas Laycock

Abstract

In this thesis my aim is to examine the common ground between formal methods and testing, and the benefits the two fields bring to one another. All too often they are regarded as mutually exclusive approaches in the development of software systems.

The thesis begins with an examination of the motivation behind software testing, a summary of its development over the past few decades, and a survey of existing techniques. This involves a detailed discussion of some of those techniques, and leads on to an extensive case study.

The case study shows how the use of a formal specification enables an existing "partition" based testing method to be used with far greater precision, but also highlights some of the limitations of the partition based techniques.

The thesis continues with a comprehensive look at the development of theoretical models of testing since the mid 1970's, and the way they have used successively more complex software models in order to be able to adequately describe suitable test cases.

The remainder of the thesis is concerned with the introduction and use of Eilenberg's X-machines as a formal model for the description of software specifications. The goal is to develop the X-machine model to the point where it is both useful and use-able as a tool for system specification, and at the same time the basis for a model of software testing so that test cases can be derived directly from the specification. To this end some of the theoretical properties of X-machines are examined, and some simple but very relevant results proved. The work is grounded on further case studies.

Acknowledgements

This work was supported by SERC Quota award number 89315080.

I would like to thank my supervisor, Mike Holcombe, for all of his help, advice and enthusiasm towards my work over the past three years.

At various stages, my work has benefited from discussions with Mike Stannett, Belkacem Mezhoud, Robert Plant, Mark Dunn and the rest of the Formal Methods and Software Engineering Group of the Computer Science Department and with Ian M^{c}Connell of the Applied and Computational Maths Department.

Thanks also to the many emacs and TEX users who have selflessly placed their work in the public domain; in particular both J. M. Spivey and Paul Taylor's LATEX styles have eased the writing and improved the appearance of this thesis.

The case study of chapter 2 would have made far less sense without the helpful comments of Martin Woodward (Department of Computer Science, Liverpool University) and the anonymous referees of of the *Journal of Software Testing, Verification and Reliability.*

Finally, thanks to Mary Rhoads who is the main reason this thesis is finished on time.

Contents

Chapter 1

Introduction

1.1 Correctness: a reason to test.

The common factor amongst system development activities is the presence of two objects—a *specification* and an *implementation*. These terms can be applied to any stage in the conventional life-cycle model of software development. For instance, for the design stage, the specification is a system specification document, and the implementation is a detailed system design. Alternatively, the terms can be applied to the development process as a whole, with the specification being the system requirements, and the implementation the system implementation.

At the highest level, the specification must accurately represent the real requirements of the system's users. This is unrealistic, but subsequent stages in the development of a system have little alternative but to assume it is true, until shown otherwise. This is true for "rapid prototyping" techniques as well as more traditional methods.

Most of the development activities concern the conversion of the specification into an implementation. But others are concerned with evaluating how well the implementation *satisfies* the specification.

The idealised goal is that the implementation satisfy the specification in every way—that the implementation can be shown to be correct with respect to the specification.

In many cases, "correct" is taken to mean equivalent, and the goal would certainly be met by a general technique for determining equivalence. However, this is impossible, since any such method would also be able to solve Turing's halting problem:

Theorem 1.1.1 There is no Turing machine E which, for any two Turing machines p_1 and p_2, has the property:

$$E(p_1, p_2) = \textit{true} \text{ iff } p_1 \text{ is equivalent to } p_2.$$

Proof (From Howden's book [27, chapter 4].)

If such a machine E exists, then it can be used to solve the halting problem. Given any machine p, use it to construct h as follows:

```
machine $h(x)$;
  $y = p(x)$; return (0);
endmachine.
```

i.e. $h(x) = 0$ if $p(x)$ terminates and doesn't terminate if $p(x)$ doesn't terminate.

Let $z(x) = 0$ for all x, and terminate for all x. Now $E(h, z)$ iff p terminates for all x. Thus knowledge of E implies knowledge of the halting problem for arbitrary p. □

1.1.1 A few definitions.

One potentially confusing issue is the variety of definitions for the related concepts of *failure*, *fault* and *error*. The following definitions broadly match with IEEE standard definitions [30].

Definition 1.1.2 Given an implementation, I, of a specification, S, a *failure* occurs if for an input, i, the output produced by the implementation is unacceptable compared to the output produced by the specification.

If the S and I are functions, then this can be written as

$$S(i) \neq I(i).$$

$\diamond$

Definition 1.1.3 Any part of the system state that could lead to failures is a *fault*. For instance a textual mistake in a program is a fault, which could lead to a number of individual failures. Until these failures actually manifest themselves, the fault is *latent*.

If I is a faulty implementation of S, this can be written as

$$S \not\equiv I.$$

$\diamond$

Definition 1.1.4 Given an input, i, and a fault, F, in an implementation, then F *affects* i if it leads to a failure when i is input. $\diamond$

Definition 1.1.5 An *error* is the direct cause of a fault. In the case of a program fault, the error is often a mistake by the programmer. Alternatively, it could be a damaged storage device, which corrupts the stored program. $\diamond$

NB The distinction between faults and errors is fine, and many authors (such as Weyuker and Ostrand [59]) refer only to "errors", but mean "faults" according to the definitions above.

Other authors (such as Laprie [34]) swap the meanings of "fault" and "error".

I shall use the definitions given here, even if discussing work where different definitions were originally used. $\diamond$

Definition 1.1.6 Given these definitions, a *correct* implementation of a specification will contain no faults. $\diamond$

A correct implementation can be achieved by either constructing it without making any errors, or by detecting and removing all the faults.

1.1.2 Correctness via proofs.

One route to correctness is to use proofs.

Attempts to prove implementations satisfy their specifications after the implementation is complete are rarely successful. Instead, a process of refinement can be used (for instance, as described by Morgan [39]). The specification, represented in some suitable formal notation is converted into an implementation using a series of simple refinements, each of which is easy to prove. In this way, there should be no faults present in the implementation.

However, there are a number of difficulties.

If the proof is constructed "by hand", there is no guarantee that there will be no errors made in constructing the proof, and so guarantee that no faults are introduced into the implementation.

To an extent, these problems can be overcome by peer reviews of the proofs involved. After all, this is the way that all classical mathematical proofs are authenticated.

Alternatively, an automatic proof system could be used to aid a human in the construction of a proof, or ultimately to perform the entire proof construction. Systems such as the BTool [1] can be used in this way. However, there is still a major drawback, in that the automatic proof system and the system of axioms used in it must be known to be correct; the tool must have been proved at some point.

In addition there must be a formal description of the environment, right down to the hardware level, and the actual physical environment must be be proved consistent with this formal model.

1.1.3 Correctness and testing.

Testing attempts to achieve correctness by detecting all the faults that are present in an implementation, so that they can be removed. In many cases the act of designing a test case that would be affected by a particular fault means that the error leading to that fault is not made when constructing the implementation, or leads to the detection of that fault without having to actually execute the implementation and observe a failure.

Testing of a system will only guarantee correctness, if the following hold:

1. The test set used is proved to be *adequate*, in that it will reveal any of the faults that could possibly occur in the implementation. The proof of adequacy must take into account the environment that the implementation is to exist in, and all of the limitations attached to proofs in general still hold.

   The most obvious way to achieve this is to include every possible input in the test set—*exhaustive* testing. But this is impractical in virtually all cases.

2. The result in each case is compared with the expected result and found to be satisfactory.

Testing and proving for correctness, as just described, are almost equally unattainable (see [17]). In practice both activities have their part to play in the production of implementations that are close to correct. In particular, since there is little prospect of eradicating all sources of errors, there will always be a justification for testing, in order to try to reveal the resulting faults.

1.2 What is testing?

The basic principle of testing for correctness is the selection of test cases that satisfy some particular *criterion*. The particular criterion used for testing computer systems have changed considerably in nature and scope as computer systems, and our knowledge of their behaviour, has developed.

1.2.1 A brief history of software testing.

A concise history of software testing is given in Gelperin & Hetzel's paper [13], which I summarise in this section. They identify the following periods.

Evolution of testing			
	-	1956	The debugging-oriented period
1957	-	1978	The demonstration-oriented period
1979	-	1982	The destruction-oriented period
1983	-	1987	The evaluation-oriented period
1988	-		The prevention-oriented period

The methods were (are) "state of the art" during the periods shown, but, of course, continue in widespread use afterwards.

Debugging-oriented testing:

Programs are written, and then simply "checked out" by the programmers until they are satisfied that all the bugs had been identified and removed. Some or all of the identification and removal activities are described as "testing"; there is no real consensus as to what the "testing" part is.

The criteria used for selecting test cases are entirely *ad hoc*, and based on the programmers' experience and their understanding of the system being built.

Demonstration-oriented testing:

Debugging and testing are identified as separate activities:

> **Debugging (or "sanity" testing)** consists of ensuring that the system runs (*i.e.* doesn't crash).
>
> **Testing** consists of ensuring that the system does what it is supposed to.

Testing is performed with the aim of showing that the system conforms to its requirements.

Destruction-oriented testing:

This view of testing gained acceptance with the publication of Myers' book *The Art of Software Testing*, [42], which described testing as

> "the process of executing a program with the intention of finding errors."

The meanings of the words *testing* and *debugging* are different again:

> **Testing** is concerned with revealing the presence of faults in the system.
>
> **Debugging** is concerned with locating and correcting those faults.

The criterion for test selection chooses test cases that reveal particular faults if they are present in the system.

Aside These last two periods separate testing out from other parts of software development, as a distinct and final stage in the life cycle model. The emphasis is on the actual execution of the test cases by the implementation of the system.

Evaluation-oriented testing:

Along with other analysis and review techniques, testing is integrated into an evaluation phase at the end of every stage of the life cycle. This period began with the publication of [43] in 1983 by the National Bureau of Standards (USA), and the realisation that the earlier in the life cycle a fault is detected, the less costly it is to correct.

For every stage in the life cycle there are requirements and products. The collective goal of the evaluation phase is to attempt to measure how well the products meet their requirements.

Prevention-oriented testing:

This approach to testing was initiated by Hetzel and Gelperin, who generalised methods for unit testing, and developed a comprehensive methodology for practical test management in [22].

The philosophy is to prevent errors in each stage of the life cycle model by using testing and other evaluation techniques as the stage progresses. (*cf.* the evaluation period, where testing and other evaluation techniques were only used at the end of each life cycle phase.) The criterion is now directed at finding places where errors might be made.

1.2.2 Summary.

Summarising from the above history, the following are the most important characteristics for the test case selection criterion to have:

- Test cases should be fault revealing. These test cases will give results contradicting the requirements if faults are present.
- Testing should be integrated into the life cycle model, so that each phase generates its own tests. The effort needed to produce test cases during each phase will be less than the effort needed to produce one huge set of test cases of equal effectiveness in a separate life cycle phase, just for testing.

There are a lots of existing testing methods, and almost as many ways of classifying them. One classification is into *program based* techniques, and *functional techniques.*

1.3 Program based testing.

These techniques are also known as *structural* and *white-box* testing.

1.3.1 Basic Principles.

Program based testing methods base their test case selection criterion on the structure of the finished code. There is a well defined hierarchy of criteria, which are described here, in ascending order of strength (see Ntafos' paper [45]):

Statement (or segment) coverage: If the test set causes every statement of the code to be executed at least once, then statement coverage is achieved.

A segment is an indivisible piece of code; no part of it can be executed without all of it being executed, *i.e.* a piece of code with no branch statements.

Branch coverage: If the test set causes every branch to be executed at least once, then branch coverage is achieved. In other words, for every branch statement, each of the possibilities must be performed on at least one occasion.

Path testing: If the test set causes every distinct execution path to be taken at some point, then path coverage is achieved. *E.g*, in the case of a loop, there are paths for each number of iterations of the loop. Even for quite short and simple programs, this level of coverage can be infeasible.

In between these coverage levels, there are all manner of other coverage measures, designed to approach path coverage without being infeasible. Two examples are:

Boundary-interior path coverage:

An overview of this technique is given by Ntafos in [45].

The number of paths through each loop is limited as follows. For each loop, identify these classes of path:

1. paths which enter the loop but don't iterate it (these are boundary paths for the loop);
2. paths which enter the loop and iterate it at least once (these are interior paths for the loop).

For class 1, perform those paths which take different paths through the loop. For class 2, perform those paths which take different paths through the loop on the first iteration.

Data-flow analysis techniques:

These techniques (described by Ntafos [45], and in more depth by Howden [27, chapter 5]) examine definitions of program variables and the subsequent use of these variables. The idea is to test all the statements with a data-flow relationship, on at least one occasion.

Suppose statement s_1 assigns a value to x, which statement s_2 then uses. Then s_1 and s_2 have a data-flow relationship. There should be a test involving the execution of s_1 followed, at some stage by the execution of s_2.

There are many variations on this theme. Some extend it to whole chains of definition-reference pairs, k-dr chains, where every chain of length k must be executed by at least one test case.

Others variations differentiate between different types of variable use: predicate use (p-use), as in branch statements, and computation use (c-use), as in the right hand side of an assignment statement. The test set must then satisfy a condition on these p and c uses, such as all c-uses, some p-uses.

1.3.2 Limitations of program based testing.

None of these program based methods use the requirements of the system in their test selection criterion. Instead, they all make the assumption that the implementation matches the requirements in its broad structure. This can be a very severe limitation if you consider the ultimate goal of testing, which is to compare the implementation with its requirements.

Errors corresponding to *missing* paths in the code will not generally be detected.

Weyuker, in [58], has given a list of properties and axioms for use in the evaluation of program-based test selection criteria. Despite being incomplete, the list of properties cannot be fulfilled by many of the program-based test selection criteria.

Another drawback of program based testing is that you have to wait until there is some of the actual code before you can even begin to construct tests. This is unsurprising given the techniques' origins in the demonstration and destruction oriented eras of testing. Testing was then carried out in its own phase of the software life cycle. More modern approaches call for testing to be integrated into all of the life cycle phases.

Nevertheless, program based testing methods are still in widespread use (see Gelperin & Hetzel [13] or one of the testing standards, such as [2]), and undoubtedly reveal a great many errors that might otherwise escape.

Secondly, the coverage levels provide a good measure of the effectiveness of tests generated in some other way. If the criterion selects test cases that do not achieve, say, statement coverage, then the criterion is probably inadequate.

1.3.3 Automation of program-based testing.

Program based testing provides a lot of scope for automating the testing procedure. This can be simple coverage analysers that are used whilst the testing is carried out, and report on the degree of coverage achieved by the test set. Other tools are a great deal more sophisticated. For instance, Roper & Smith [52], built a tool that accepts the detailed design of the program in the form of a Jackson Structure diagram, and produces test sets suitable for use on the actual code. This is particularly interesting, as it highlights the need for there to be something to compare the implementation with, in this case a JSP design.

1.3.4 Mutation testing.

Mutation testing (see Woodward's summary [60]) is based around making large numbers of minor changes to the implementation under test. Each modified piece of code is a *mutant.* A test set is applied to the mutants, and the output compared to that from the original. For a given mutant, if the output from the test differs from that from the original, then the mutant has been *killed.* Those mutants indistinguishable from the original by the test set are said to be *live.* If there is a live mutant after a test, there are two possibilities:

- the test was not good enough; an improved version should be devised that kills the mutant, or reveals the original to contain a fault;
- the mutant is, in fact, equivalent to the original.

There are several versions of mutation testing, according to how the mutants are generated.

Strong mutation testing, as described by DeMillo *et al.* in [6], involves the systematic mutation of all the operators in a program, and the running of the complete test set on each mutant. This is computationally expensive, so, in some cases, restricted subsets of the operators are mutated instead.

Weak mutation testing, proposed by Howden [26], aims to cut down on the computational cost, by combining several mutants into a single new version of the program. In this way, it is not necessary to run the complete test set for every mutant. However, there is a risk that mutants will "cancel one another out".

Firm mutation testing, suggested by Wu *et al.* in [61], is an intermediate strategy. It takes advantage of an interactive development environment to allow code fragments to be mutated and executed in partial isolation from the rest of the code.

1.4 Functional Testing.

These methods are also known as *black-box* methods.

Functional testing methods base their test case selection criteria largely on the intended functionality of the implementation, *i.e.* on the specification, or requirements. This fits in well with the goal of comparing implementations with their requirements.

1.4.1 The Category-Partition method.

A large number of functional test methods are based on the idea of partitioning the input domain. These methods have a great deal in common, so I will only discuss one in detail, the category-partition method.

This method was presented by Ostrand and Balcer [46]. It was designed to be used in conjunction with a tool that they had developed. The required tests are described using the Test Specification Language, and the tool then generates *test frames* which describe individual test cases.

As in almost all such methods, there is the assumption that the requirements and specification are informal and presented in natural language. There are several steps to the method.

Analysis of the specification:

Identify functional units that can be individually tested; either top level user commands or functions that are called by them, or lower level functions. Several stages of decomposition may be required.

For all the functions identified, find the parameters (*i.e.* explicit inputs to the functional unit, either by the program or by the user) and environment conditions (*i.e.* characteristics of the system state at the time the function is executed) that affect the function's behaviour.

Example: Consider the specification of a sorting program.

> The program is to accept an array of variable length and containing items of arbitrary type. The output is to be a permutation of the input array, but with the values correctly ordered (with respect to a total ordering). Further, the program is to output the maximum and minimum elements in the array.

Then, the only parameters are the unsorted array as input, the sorted array as output, and the values of the maximum and minimum elements respectively.

Categorise the parameters and environment conditions:

For each parameter and environment condition, identify properties and characteristics that have particular effects on the function's behaviour. Classify the characteristics of the parameters and environment conditions into *categories* that characterise the behaviour of the function. This is best illustrated by example.

This process can uncover many ambiguities and mistakes in the original specification.

Example: For the sort program, the categories are as follows:

- the array's size;
- the type of the elements;
- the value of the maximum in the array;
- the value of the minimum in the array;
- the positions of these values in the unsorted array;
- the positions of these values in the sorted array;
- the order of the unsorted array.

Partition the categories into *choices*:

Determine the different significant cases that can occur within each parameter or environment condition category. These cases are *choices*. Each choice consists of a subset of the category's values, which will all lead to the same sort of behaviour. The choices must be mutually exclusive.

The partitioning is based on the specification, the implementation, or any other design documents that are available, and the tester's past experience of selecting test cases.

Example: Using the "position of maximum element of the unsorted array" partition from above, the choices might be for:

- maximum element at first position in the unsorted array;
- maximum element at last position of the unsorted array;
- maximum element somewhere in the middle of the unsorted array;
- more than one element with the maximum value in various positions in the unsorted array.

```
TestSpec      ::= ⟨FunctionSpec⟩
FunctionSpec  ::= Function : FunctionName\n
                      ParamSpec
                      {EnvSpec}
                  EndFunction; \n
ParamSpec     ::= Parameters : \n
                      ⟨AParameter⟩
EnvSpec       ::= Environments : \n
                      ⟨AnEnvironment⟩
AParameter    ::= Number ParamName\n
                      ⟨Choice⟩
AnEnvironment ::= Number EnvironmentName\n
                      ⟨Choice⟩
Choice        ::= • ChoiceName{{[if Condition]}{[property PropertyName]}}
                          {[single] | [error]}\n
```

⟨. . .⟩ indicates repetition (at least once) of an item
{. . .} indicates an optional item
. . . | . . . indicates alternative items
`this font` indicates literal text, and
`\n` indicates a (literal) new line.

Figure 1.1: BNF syntax for TSL

Determine *constraints* among the choices:

Decide what effect choices from one category will have on those from another. Look for mutual exclusion, special restrictions and so on.

Also, mark any choices that are expected to generate an error with `[error]`. Further, choices that are, in some way, special or redundant can be marked `[single]` (this should be done very carefully). Both these marks will cause the test frame generator to produce only simple test frames for these choices—they need not be combined with all the other choices.

Example: If the array size partition were chosen to be one, then there can be only one maximum element in the array; there is no point trying to devise tests in which there are more than one element with the maximum value.

Write and process the test specification:

The test specification consists of the categories, the choices within the categories and the constraints on the choices. These are prepared in a standard format, the Test Specification Language (TSL), and fed into a test frame generation tool, which then generates test frames (*i.e.* a set of choices from the test specification; each category supplies either exactly one or none of its choices) for that functional unit. Ostrand & Balcer did not precisely describe the TSL; in figure 1.1 I introduce a syntax that is consistent with the way TSL is used in [46].

Examine the generated test frames:

The test frames generated by the tool are evaluated. If they are unsatisfactory, go back to the constraint determining step. Unsatisfactory could mean any of the following:

- There are some test frames that are obviously missing.
- There are some test frames that are obviously impossible.
- There are far too many or far too few test frames.

Transform the test frames into test scripts:

Convert each test frame into an actual test case. Do this by selecting an actual value from each of the choices in the test frame. At the same time calculate the expected output for the test case. Organise these cases into scripts, suitable for execution by the implementation.

Advantages of the method.

1. The test specifications are given in a concise and uniform way, which is valuable for quality analysis activities, and is often required by test standards.
2. The process of working through all the details of the method may well reveal limitations of the design specification. If so, these should of course be addressed, and then the implications considered for the parts of the testing process already carried out.
3. The test specifications can be started early in the development process.
4. The test specifications can be easily modified as the system evolves.
5. The number of tests can be controlled in a relatively reliable way.
6. The test frame generation tool eliminates many of the possibilities for human error in the testing procedure.

Limitations of the method.

1. It is difficult to describe the early stages of the method formally. The test selection criterion is that every possible combination of unconstrained choices is represented by some particular test frame. Each test frame must have exactly one or none of the choices from each category. However, any further formal analysis is hampered by the lack of a formal definition of either categories or the partition of categories into choices.
2. As a result, the method relies heavily on the experience of the tester. This could lead to non-uniform tests.
3. Although work can start on the testing at an early stage, none of the tests can actually be carried out until there is a completed version of the implementation.

1.4.2 Other partitioning methods.

Several other functional testing methods are broadly similar to the category-partition method just discused. The basic principle has doubtless been used in an *ad hoc* manner for as long as systems have been developed.

Condition tables.

One of the first techniques to be documented, this was presented by Goodenough and Gerhart [17], at the same time as their theoretical basis of testing (which is discussed in chapter 3), so it is also one of the most formally developed methods. It explicitly links the theory with the concept of correctness; they show that it is just as difficult to guarantee correctness via testing as via proof.

Instead of categories, there are *conditions*, which determine the behaviour of the system. Instead of partitioning into choices, the possible *values* that the condition could take are considered.

This information is laid out in a table, with a row for each condition, and a column for each possible combination of values. Thus each column corresponds to a test frame.

There is a limited use of constraints between conditions, but this is only to indicate when they are mutually exclusive. There is no way to reduce an overly large set of test cases by putting in some extra restraints.

Revealing subdomains.

This was suggested by Weyuker and Ostrand [59]. They point out some limitations in the theory of Goodenough and Gerhart [17], and some of the difficulties of applying it to real systems. They develop this new method and extend the theory.

The idea is to split the input domain of the program into *revealing subdomains*. All of the elements in a revealing subdomain will either be processed correctly or incorrectly, and so only one element from the subdomain is needed as a test case. As it stands, this is just as impractical as a proof. So the subdomains need only be revealing with respect to a particular fault. This is very similar to the situation where you have found the categories of a function and partitioned them into choices.

Cause-Effect Graphing.

Elmendorf [11] introduced this method, but Myers illustrated it, and brought it to wider attention [40, 42].

Choose the function that is to be tested. Identify all the implicit and explicit *causes* (input conditions) and *effects* (output actions) for the function, using the specification. (*cf.* identifying categories and splitting them into choices in the Category-Partition method.) Next, build the graph. Causes and effects are nodes in the graph, linked by arcs representing relationships between causes and effects. For instance, if some of the causes must all be present in order for a particular effect to occur, there are arcs from the causes to the effect, labelled with an AND. Similarly, arcs can be labelled with OR or NOT. If the relationships are particularly complicated, intermediate nodes can be introduced.

Once the graph has been built, construct a decision table. For each effect, find all the different combinations of causes that lead to it. Each of these will form a test frame. At the same time, list the states of the other effects for each of the combinations of causes. This gives you information on the expected output for each of the frames.

This method is criticised by Ostrand & Balcer for the complexity of the graphs produced, and the difficulty of altering them after they have been constructed. However, with a suitable tool for constructing and editing such a graph, this method would become quite practical.

Limitations of these "partitioning" methods.

All of these methods attempt to split the input domain of a function or program into subsets the elements of which will behave in a broadly similar fashion. The assumption is that the presence of a fault will affect every element of a subset. This is intuitively appealing and meets with some success in practice. However, the partitioning process is difficult to describe formally, and so it is hard to assess the criteria for their adequacy.

Also, see section 1.5 on the confidence building properties of partition testing.

1.4.3 Other functional methods.

All of the preceding methods are directed largely at dynamically testing actual code. Given that modern quality standards require testing throughout the development process (see [44, 13]) there is a need for some higher level testing methods.

Testing specification refinements.

Some work has been done in the area of using formal function definitions for testing purposes. There is a split between those using model type specifications, like Z (Hayes [21]), and those using axiom based specifications, like OBJ (Gerrard *et al.* [15]).

The general idea is to use the pre, post and invariant conditions of the specification, which would not normally be used other than for a proof, for testing purposes. For instance, the implementation of a simple symbol table might be as an ordered list of symbols. The specification describes this using an invariant condition called ORDERED. The ORDERED condition is of no consequence to the end user. However, by implementing some code to check the ORDERED condition, tests can be carried out to see if other operations on the symbol table violate the invariant.

OBJ based specifications have the advantage that they can be directly executed, so that conditions such as ORDERED can be confirmed at the specification stage. With Z, there is at present no way to execute a specification, and the ORDERED condition would have to be implemented by hand, along with prototype code for the operations on the symbol table.

Functional tests from JSP.

In section 1.3.3, the work of Roper and Smith in producing tests from JSP diagrams was mentioned. They developed this, in [53], into a functional testing method, based on the specification. By restricting the functions used to five basic function types (data access, data storage, arithmetic expression, arithmetic relation and Boolean expression), the specification can be made concise and unambiguous in an operational specification. Also rigorous tests for these five types of function are described in Howden's book, [27], and so tests can be derived directly from the operational specification.

The tests derived from the operational specification are applied to the JSP program design, and to the actual code produced from the JSP.

1.4.4 Completeness of a specification.

Given a specification, it is desirable if it is *consistent* and *complete*. Loosely, this means that the specification must be unambiguous and be defined for all possible inputs. In [31], P. Jalote describes a method for testing specifications given in the OBJ language for completeness. OBJ constructs the specification of operations on abstract data types axiomatically, and Jalote only considers incompleteness caused by missing axioms.

A tool is used to generate the tests automatically. The tests are based on the syntax part of the specification, which gives the signatures of the operations. (There is no point basing them on the semantics, since that is where the errors leading to incompleteness will be.) The tool generates all of the syntactically possible expressions down to a certain depth of operation applications. The test cases are these expressions, with the various output operations applied to them.

Jalote has found the method to work well in practice, although there are limitations on the axioms that it can cope with.

Aside Mutation testing can also be applied to an executable specification. Woodward outlines the approach in [60]. ◇

1.5 Statistical testing and reliability.

So far, I have only discussed testing methods aimed at fault detection, with the goal of correctness in mind. This is not the only motivation for testing.

Suppose that a system has been extensively tested without revealing any failures. These tests mean that there is a higher level of *confidence* in the system, (or a reduced expectation of failure) than before they were carried out.

Statistical methods can be used to try to quantify the increase in confidence by estimating the *probability of failure*. There are a number of different statistical models used (Miller *et al.* [37], Hamlet & Taylor [20], Weiss & Weyuker [57]), and they lead to conflicting claims as to the benefits of different types of testing. In particular, Hamlet and Taylor claim that "partition testing does not inspire confidence," whereas Miller *et al.* describe circumstances where partitioning can increase confidence.

Tests are selected randomly using a probability density function based on the operational input distribution. Then, each test that does not lead to a failure slightly reduces the estimated probability of a failure occurring. The extent to which it does this depends on the particular model used, and the assumptions made about the software's behaviour.

Chapter 2

Case study: Z specification and testing.

2.1 Overview

This chapter has two aims:

- To examine the use of a partition based testing method (Ostrand & Balcer's category-partition method [46]) in detail.
- To demonstrate the benefits of using a formal specification (written in Z, see Spivey's *Reference Manual* [55]) in the development of those tests.

An extended version of this chapter is published as [35].

The system described is a dating agency, originally inspired by an example of Howden's [27, chapter 2]. Details of users are recorded in a data-base (although details of how the data-base is managed are kept to a minimum), and the specification here covers queries made to the data-base for suitable partners for a client.

2.2 Specification for a dating system in Z.

2.2.1 Basic abstract definition

To begin with introduce some basic types, representing users and their details in the system.

$$[\mathit{USER}, \mathit{USERRECORD}]$$

In practice, these types would have some internal structure (for instance, an element of *USERRECORD* would contain details of a *USER*'s likes and dislikes plus other relevant data), but for the moment this is not developed.

Now define the data-base that stores information on all the dating system's users:

$$\mathit{DateFile} == \mathit{USER} \mathrel{⇸} \mathit{USERRECORD}$$

$$\begin{array}{|l}
\mathit{DF} \\
\hline
\mathit{dtfile} : \mathit{DateFile} \\
\hline
\end{array}$$

In other words *DF* is a structure consisting of a partial function from *USER*s to *USERRECORD*s called *dtfile*. This means that the names of actual users, of type *USER* must be unique. In other words, treating *DF* as a database, elements of *USER* are used as the key field.

Next introduce a function to perform compatibility analysis:

$$\mathit{compatible} : \mathit{USER} \rightarrow (\mathit{USER} \rightarrow \mathbb{N})$$

This function compares two user records and returns a numerical indication of their compatibility; the greater the number returned, the greater the compatibility. The details of how this is achieved depend on the internal structure of *USERRECORD* and are deferred until that structure is defined

This schema is an example of function Currying. Given a single argument of type *USER*, *compatible* will return a function of type $\mathit{USER} \rightarrow \mathbb{N}$. Given a pair of arguments, of types *USER* and *USER*, it returns an element of type $\mathbb{N}$. Strictly speaking, only the first argument is used by *compatible*, the second argument being given to the function returned, and so typical applications of function might look like this:

$\mathit{compatible}(u_1)$ which is of type $\mathit{USER} \rightarrow \mathbb{N}$
or
$(\mathit{compatible}(u_1))(u_2)$ which is of type $\mathbb{N}$.

However, for convenience, the notation $\mathit{compatible}(u_1, u_2)$ will be taken to be equivalent to (the strictly correct) $(\mathit{compatible}(u_1))(u_2)$.

It is now possible to specify what it means for a date to be "best".

$$\mathit{bestdate} : \mathit{DateFile} \rightarrow (\mathit{USER} \rightarrow \mathit{USER})$$

$$\begin{aligned}
&\forall\, \mathit{dtfile} : \mathit{DateFile};\ \forall\, \mathit{client} : \mathit{USER} \bullet \\
&\quad \mathit{bestdate}(\mathit{dtfile}, \mathit{client}) \in \operatorname{dom} \mathit{dtfile} - \{\mathit{client}\} \wedge \\
&\quad \mathit{compatible}((\mathit{bestdate}(\mathit{dtfile}, \mathit{client})), \mathit{client}) \\
&\qquad = \mathit{max}(\operatorname{ran}(\mathit{compatible}(\mathit{client})))
\end{aligned}$$

In other words, the *bestdate* function is simply defined by considering users not including the *client* who's compatibility with the *client* is maximal. Furthermore, the *bestdate* function is defined non-deterministically, in that if there is more than one user with maximal compatibility, the specification does not define *which* of them is returned.

So a system operation can be defined that finds the best date for a client:

BestDate

$$\begin{aligned}
&\Xi DF \\
&\mathit{client}? : \mathit{USER} \\
&\mathit{bestdate}! : \mathit{USER} \\
\hline
&\mathit{bestdate}! = \mathit{bestdate}(\mathit{dtfile}, \mathit{client}?)
\end{aligned}$$

2.2.2 Basic implementation details

Given the above abstract description, consider a possible means of actually implementing it.

Suppose the *dtfile* has been implemented using an ordered sequence, with respect to some ordering $<_{USER}$ on *USER* (the definition of which is deferred). The specification of this implementation is as follows:

$$\mathit{RecPair} ::= \mathit{USER} \times \mathit{USERRECORD}$$

$$
\begin{array}{|l}
\textit{DFImp} \\
\hline
\textit{dfseq} : \mathrm{seq}\, \textit{RecPair} \\
\hline
\textit{ordered}(\textit{dfseq}) \\
\hline
\end{array}
$$

where

$$
\begin{array}{|l}
\textit{ordered} : \mathrm{seq}\, \textit{RecPair} \rightarrow \textsc{Boolean} \\
\hline
\forall\, s : \mathrm{seq}\, \textit{RecPair};\ i, j : \mathrm{dom}(s) \bullet i < j \Rightarrow s(i).\textit{user} <_{\textit{USER}} s(j).\textit{user}
\end{array}
$$

This schema sees the use of $.\textit{user}$ as a selector function. The elements $s(i)$ are of type *RecPair*, and $s(i).\textit{user}$ refers simply to the part of the tuple of type *USER*. Similarly, $s(i).\textit{userrecord}$ would refer to just the *USERRECORD* part of the tuple.

Before continuing with the implementation consider its relationship with the earlier abstract specification. It is important that this is well defined.

$$
\begin{array}{|l}
\textit{DF} - \textit{DFImp} \\
\hline
\textit{DF} \\
\textit{DFImp} \\
\hline
\textit{DF} = \{\textit{Pr} : \mathrm{ran}(\textit{dfseq}) \bullet \textit{Pr}.\textit{user} \mapsto \textit{Pr}.\textit{userrecord}\} \\
\hline
\end{array}
$$

In other words, a pair in the ordered list of *DFImp* corresponds to a maplet ($\textit{A\_User} \mapsto \textit{A\_Record}$) in *DF*, where *A_User* and *A_Record* are elements of type *USER* and *USERRECORD* respectively.

Now develop implementation definitions that allow the best date for a client to be found in *DFImp*, that conform to the abstract definition given in *DF*.

A lookup function for locating the details of a particular user is required:

$$
\begin{array}{|l}
\textit{lookup} : \textit{USER} \times \mathrm{seq}\, \textit{RecPair} \nrightarrow \textit{RecPair} \cup \{\textit{null}\} \\
\hline
(\forall\, \textit{usr} : \textit{USER};\ \textit{file} : \mathrm{seq}\, \textit{RecPair} \bullet \\
\quad (\exists\, \textit{rec} : \textit{USERRECORD} \bullet (\textit{usr}, \textit{rec}) \in \mathrm{ran}\, \textit{file} \\
\quad\quad \land\ \textit{lookup}(\textit{usr}, \textit{file}) = (\textit{usr}, \textit{rec})) \\
\quad \lor \\
\quad (\neg\ \exists\, \textit{rec} : \textit{USERRECORD} \bullet (\textit{usr}, \textit{rec}) \in \mathrm{ran}\, \textit{file} \\
\quad\quad \land\ \textit{lookup}(\textit{usr}, \textit{file}) = \textit{null})
\end{array}
$$

It is necessary to be able to search a sequence for the best date

$$
\begin{array}{|l}
\textit{BestInList} : \textit{USER} \times \mathrm{seq}\, \textit{RecPair} \rightarrow \textit{RecPair} \\
\textit{RcvBestInList} : \textit{RecPair} \times \mathrm{seq}\, \textit{RecPair} \rightarrow \textit{RecPair} \\
\hline
\forall\, \textit{client} : \textit{USER};\ \textit{dfile} : \mathrm{seq}\, \textit{RecPair} \bullet \\
\quad \textit{BestInList}(\textit{client}, \textit{dfile}) = \textit{RcvBestInList}(\textit{lookup}(\textit{client}, \textit{dfile}), \textit{dfile}) \\
\forall\, \textit{clinfo} : \textit{RecPair};\ \textit{file} : \mathrm{seq}\, \textit{RecPair} \bullet \\
(\textit{head}(\textit{file}) \neq \textit{clinfo} \land \\
\quad \textit{Better}(\textit{head}(\textit{file}), \textit{RcvBestInList}(\textit{clinfo}, \textit{tail}(\textit{file})), \textit{clinfo}) \\
\quad\quad \Rightarrow \textit{RcvBestInList}(\textit{clinfo}, \textit{file}) = \textit{head}(\textit{file})) \\
\lor \\
((\textit{head}(\textit{file}) = \textit{clinfo} \\
\quad \lor\ \textit{Better}(\textit{RcvBestInList}(\textit{clinfo}, \textit{tail}(\textit{file})), \textit{head}(\textit{file}), \textit{clinfo})) \\
\quad\quad \Rightarrow \textit{RcvBestInList}(\textit{clinfo}, \textit{file}) = \textit{RcvBestInList}(\textit{clinfo}, \textit{tail}(\textit{file})))
\end{array}
$$

where

$$\begin{array}{|l}
Better : RecPair \times RecPair \times RecPair \rightarrow \textsc{Boolean} \\
\hline
Better(x, y, z) = true \Leftrightarrow x \text{ is a better match for } z \text{ than } y
\end{array}$$

The actual definition of this is left, pending the definition of the internal structure of *USERRECORD*.

The function *RcvBestInList*, which does the bulk of the work, is defined recursively on sequences of *RecPairs*. At each level of recursion, it simply uses *Better* to decide if the first *RecPair* of the sequence is a better match for the client user than the result of applying *RcvBestInList* to the remainder of the sequence (*i.e.* the result of finding the best date for the client user in the remainder of the sequence).

Notice that the non-determinism from the specification of *bestdate* has been resolved in the specification of *BestInList*; the last user in the sequence with maximal compatibility will be selected as the value of *BestInList*. This is an arbitrary solution to the problem, and the simplest rather than the one that would be most useful in practice.

This leads to the implementation for giving the best date:

$$\begin{array}{|l}
\textit{BestDateImp} \\
\hline
\Xi DFImp \\
client?, date! : USER \\
\hline
date! = BestInList(client?, dfseq).user \\
\hline
\end{array}$$

In practice, it would be important to show that *BestDateImp* satisfies *BestDate*, in that the *date*! chosen by *BestDateImp* satisfies the condition given in *BestDate*. This is omitted here for the sake of brevity.

As a final stage to specifying the system, add some exception handling operations:

Introduce a new type

$$REPORT ::= Ok \mid Error$$

and some new schemas

$$\begin{array}{|l}
\textit{Success} \\
\hline
rep! : REPORT \\
\hline
rep! = Ok \\
\hline
\end{array}$$

AbstractFailure

$$\begin{array}{l}
\Xi DF \\
client?, date! : USER \\
rep! : REPORT \\
\hline
(client? \notin \mathrm{dom}(dtfile) \lor \mathrm{dom}(\{client?\} \lhd dtfile) = \varnothing \lor client? = nil) \\
\land\ rep! = Error
\end{array}$$

ImpFailure

$$\begin{array}{l}
\Xi DFImp \\
client? : USER \\
rep! : REPORT \\
\hline
rep! = Error \land ((dfseq = <>) \lor\ (lookup(client?, dfseq) = null) \lor (client? = nil))
\end{array}$$

Now define a *RobustBestDate*

$$RobustBestDate == AbstractFailure \lor (BestDate \land Success)$$

and a *RobustBestDateImp*

$$RobustBestDateImp == ImpFailure \lor (BestDateImp \land Success)$$

The operation *RobustBestDateImp* is the one that is going to be tested.

Before going on to consider testing it, it remains to fill in the details of the internal structure of *USERRECORD*, and some of the other definitions dependent on them.

$$\begin{array}{l}
[CHAR] \\
USER ::= \mathrm{seq}\, CHAR \\
SEX ::= nulsex \mid male \mid female \\
PIZZA ::= nulpz \mid LikesPizza \mid DoesntLikePizza \\
USERRECORD ::= nulUR \mid UR\langle\langle SEX \times PIZZA\rangle\rangle
\end{array}$$

In other words, the elements of *USER* are simple strings of characters, and *USERRECORD*s are records made up from a *SEX* field and a *PIZZA* field (this is a fairly simple model of a dating agency). Any or all of these can take null values.

$$\begin{array}{l}
Better : RecPair \times RecPair \times RecPair \rightarrow \mathrm{BOOLEAN} \\
compatibility : USERRECORD \times USERRECORD \rightarrow \mathbb{N} \\
\hline
\forall r_1, r_2, r_3 : RecPair \bullet \\
\quad Better(r_1, r_2, r_3) \Leftrightarrow \\
\qquad compatibility((r_1).userrecord, (r_3).userrecord) > \\
\qquad\quad compatibility((r_2).userrecord, (r_3).userrecord) \land \\
\qquad nulUR \notin \{(r_i).userrecord \mid i = 1, 2, 3\} \land \\
\qquad nulsex \notin \{((r_i).userrecord).sex \mid i = 1, 2, 3\} \land \\
\qquad nulpz \notin \{((r_i).userrecord).pizza \mid i = 1, 2, 3\} \\
\forall ur_1, ur_2 : USERRECORD \bullet \\
\quad (ur_1).sex = (ur_2).sex \Rightarrow compatibility(ur_1, ur_2) = 0 \\
\quad (ur_1).sex \neq (ur_2).sex \land (ur_1).pizza \neq (ur_2).pizza \Rightarrow \\
\qquad compatibility(ur_1, ur_2) = 1 \\
\quad (ur_1).sex \neq (ur_2).sex \land (ur_1).pizza = (ur_2).pizza \Rightarrow \\
\qquad compatibility(ur_1, ur_2) = 2
\end{array}$$

This defines *Better* in terms of *compatibility*, which simply gives a score between 0 and 2 depending on whether the two *USERRECORD*s supplied are different sexes and both like pizza.

Once again projection functions are used to access individual parts of the tuples. For instance, $((r_i).userrecord).pizza$ refers to the *PIZZA* part of the *USERRECORD* part of r_i which is of type *RecPair*.

Finally, the *compatible* function used earlier can now be properly defined:

$$
\begin{array}{|l}
compatible : USER \rightarrow (USER \rightarrow \mathbb{N}) \\
\Xi DF \\
\hline
\forall\, u_1, u_2 : USER \\
\qquad compatible(u_1, u_2) = compatibility(dtfile(u_1), dtfile(u_2))
\end{array}
$$

2.3 Application of Category Partition Method.

The functions are *BestInList*, *RcvBestInList*, *Better*, *lookup*.

2.3.1 Identify ***parameters***, and ***environment conditions***:

The distinction between parameters and environment conditions is hazy at best, particularly if a functional approach to specification is taken. So, the distinction is ignored here, and a simple combined list for each function is given.

lookup: one of type *USER*, one of type *RecPair* and one of type *dfseq* (the file of system users).

Better: 3 *RecPair*s the first 2 being compared to see which is more compatible with the 3rd, and a boolean for returning the answer.

BestInList: one of type *USER*, a *dfseq* (*i.e.* a file), and a *RecPair* corresponding to the best date for the user in the *dfseq*.

RcvBestInList: one of type *RecPair*, a seq *RecPair* (*i.e.* a file), and another *RecPair* corresponding to the best date in the list for the first *RecPair*.

2.3.2 Split the parameters into characteristic ***categories***

lookup: the *USER* being looked up, the *dfseq* (file) being looked in, the length of the *dfseq*, the number of occurrences of the *USER* in the *dfseq*, the position of the *USER* in the *dfseq* (if its there at all).

Better: the first *RecPair*, the second *RecPair*, the third *RecPair*, the relative *compatibility* of the first and second with the third.

BestInList: the *USER*, the *dfseq* (file), the length of the *dfseq*, the position and number of occurrences of the *USER* in the *dfseq*, the *bestdate* for the *USER* in the *dfseq*, the position and number of occurrences of the *bestdate* in the *dfseq*.

RcvBestInList: the *RecPair* of the client *USER*, the seq *RecPair*, the length of the seq *RecPair*, the position and number of occurrences of the client *USER* in the seq *RecPair*, the position and number of occurrences of potential *bestdate*s in the seq *RecPair*.

2.3.3 *Partition* the categories into *choices*

lookup:

1. the *USER* being looked up can be chosen from {missing, present};
2. the *dfseq* being looked into can be {missing, present};
3. the *length* of the *dfseq* can be {zero, singleton, longer};
4. the *number* of occurrences of the *USER* in the *dfseq* can be {never, once, more_than_once};
5. the *position* of the *USER* in the *dfseq* can be {not_there, first, last, middle}.

Better:

1. the first, second and third *RecPair*s can be {present, missing};
2. the relative compatibility of the first and second *RecPair*s can be {first_better, second_better, equal}.

BestInList:

1. the *USER* can be {missing, present};
2. the *dfseq* can be {missing, present};
3. the *length* of the *dfseq* can be {zero, singleton, double, longer};
4. the *number of occurrences* of the *USER* in the *dfseq* can be {zero, once};
5. the *position* of the *USER* in the *dfseq* can be {not_there, start, end, middle};
6. the *number of occurrences* of potential *bestdate*s can be {zero, once, more_than_once};
7. the *positions* of potential *bestdate*s can be {not_there, start_only, end_only, middle_only, start_middle, start_end, middle_end, start_middle_end}.

RcvBestInList:

1. the *RecPair* of the *client* can be {missing, present};
2. the seq *RecPair* being searched can be {missing, present};
3. the *length* of the seq *RecPair* can be {zero, singleton, double, longer};
4. the *number of occurrences* of the client *RecPair* in the seq *RecPair* can be {zero, once};
5. the *position* of the client *RecPair* in the seq *RecPair* can be {not_there, start, end, middle};
6. the *number of occurrences* of potential *bestdate*s can be {zero, once, more_than_once};
7. the *position* of potential *bestdate*s can be {not_there, start_only, end_only, middle_only, end_only, start_middle start_end, middle_end, start_middle_end}.

A parameter is "missing" if it is not supplied to the function, or if it takes a *null* value of some sort.

2.3.4 Pause for breath

Before going on to simulate the application of a test generating tool to the above lists of choices, it is beneficial to re-assess their usefulness. This is prompted by observing the similarity between parts of the above lists. In particular, those for *BestInList* and *RcvBestInList* are almost exactly alike, and that for *lookup* is duplicated almost exactly in both *BestInList* and *RcvBestInList*. As things stand, the tests for (say) *lookup* will be repeated 3 times, which seems very wasteful. Two approaches suggest themselves:

1. Only test the function with the "biggest" list, *i.e. BestInList*. The justification is that the choice list for this "includes" the choice lists for the sub-functions (*i.e. lookup*, *Better* and *RcvBestInList*). If the sub-functions are not then tested as thoroughly as they would have been if they were considered individually, then this is assumed to be unimportant as, all of the *critical* parts of the sub-functions will be tested. This is reminiscent of the "big-bang" approach to testing where coding is completed before any testing can be performed.
2. Test all the "small" functions first, and then remove the overlap between them and the "bigger" functions. Repeat this until you get to the top-level functions. In this way, you can avoid duplicating testing effort, and start performing tests on low level functions as they are ready. However, there is the difficulty of deciding when a "small" function's constraint list is included in that of a "bigger" function. For instance, the *Better* function is used by *RcvBestInList*, but it is not immediately clear how. This is a more "bottom-up" approach, and it requires that sub-functions are test-able individually. This may require the construction of special test harness code.

A hybrid approach is adopted here, considering *BestInList* and *Better*.

2.3.5 Now determine *constraints* on the choice partitions

It is important that the order that the constraints are defined is irrelevant. Also, it is important to understand the meaning of an [if *property*]—it indicates that that partition choice *can* (but need not *necessarily*) be used if the *property* holds; the choice *cannot* be used if the *property* does not hold.

The categories, partitioned into choices, and their constraints, can be written down using Ostrand & Balcer's Test Specification Language [46], for which I introduced a formal syntax in figure 1.1 one page 14.

`Function:`*Better*
`Parameters:`

1. 1st *RecPair*:
   - present [1stPres]
   - missing [`error`]
2. 2nd *RecPair*:
   - present [2ndPres]
   - missing [`error`]
3. 3rd *RecPair*:
   - present [3rdPres]
   - missing [`error`]
4. The relative compatibility of the first and second *RecPair*s, compared with the third:
   - first_better [if 1stPres & 2ndPres & 3rdPres]
   - second_better [if 1stPres & 2ndPres & 3rdPres]
   - equal [if 1stPres & 2ndPres & 3rdPres]

`EndFunction;`

`Function:`*BestInList*
`Parameters:`

1. The *USER* for whom a date is being sought:
   - present [property IdThere]
   - missing [property NoIdThere] [`error`]

2. The *dfseq* being searched:
   - present [property SeqThere]
   - missing [if Empty] [property NoSeqThere] [`error`]
3. The length of the *dfseq*:
   - zero [property Empty]
   - singleton [if SeqThere] [property OneLong]
   - double [if SeqThere] [property TwoLong]
   - longer [if SeqThere] [property LotsLong]
4. The number of occurrences of the *USER* in the *dfseq*:
   - zero [property UserNotIn]
   - once [if IdThere & SeqThere] [property UserIn]
5. The position of the *USER* in the *dfseq*:
   - not_there [if UserNotIn]
   - start [if UserIn & (OneLong or TwoLong or LotsLong)]
   - end [if UserIn & (TwoLong or LotsLong)]
   - middle [if UserIn & LotsLong]
6. The number of occurrences of potential *bestdate*s:
   - zero [if (UserNotIn or NoIdThere or OneLong)] [property NoBestDate]
   - once [if UserIn & (TwoLong or LotsLong)] [property BestDate]
   - more_than_once [if UserIn & LotsLong] [property BestDate]
7. The positions of potential *bestdate*s:
   - not_there [if NoBestDate]
   - start_only [if BestDate]
   - middle_only [if BestDate]
   - end_only [if BestDate]
   - start_middle [if BestDate]
   - start_end [if BestDate]
   - middle_end [if BestDate]
   - start_middle_end [if BestDate]

```
EndFunction;
```

2.3.6 Apply the test generation tool to produce test frames

Sadly, no test generation tool was available, but here are simulated results of applying it to the above data.

Test frames are given in the following format:

Xi ={1.missing, 2.present, 3.present, 4.NA}

which would be test number Xi, with the particular choices from each category given. In other words, the choice for category 1 is "missing", *etc.* NA indicates that no choice from a category is applicable.

1. *Better*:

   A1 = {1.missing, 2.present, 3.present, 4.NA},

   A2 = {1.present, 2.missing, 3.present, 4.NA},

   A3 = {1.present, 2.present, 3.missing, 4.NA},

A4 = {1.missing, 2.missing, 3.present, 4.NA},
A5 = {1.missing, 2.present, 3.missing, 4.NA},
A6 = {1.present, 2.missing, 3.missing, 4.NA},
A7 = {1.missing, 2.missing, 3.missing, 4.NA},
A8 = {1.present, 2.present, 3.present, 4.first_better},
A9 = {1.present, 2.present, 3.present, 4.second_better},
A10 = {1.present, 2.present, 3.present, 4.equal}.

2. *BestInList*:

B1 = {1.missing, 2.missing, 3.zero, 4.zero, 5.not_there, 6.zero, 7.not_there},
B2 = {1.missing, 2.present, 3.NA, 4.zero, 5.not_there, 6.zero, 7.not_there},
B3 = {1.present, 2.missing, 3.zero, 4.zero, 5.not_there, 6.zero, 7.not_there},
B4 = {1.present, 2.present, 3.zero, 4.zero, 5.not_there, 6.zero, 7.not_there},
B5 = {1.present, 2.present, 3.singleton, 4.zero, 5.not_there, 6.zero, 7.not_there},
B6 = {1.present, 2.present, 3.singleton, 4.once, 5.start, 6.zero, 7.not_there},
B7 = {1.present, 2.present, 3.double, 4.zero, 5.not_there, 6.zero, 7.not_there},
B8 = {1.present, 2.present, 3.double, 4.once, 5.start, 6.zero, 7.not_there},
B9 = {1.present, 2.present, 3.double, 4.once, 5.start, 6.once, 7.end},
B10 = {1.present, 2.present, 3.double, 4.once, 5.end, 6.zero, 7.not_there},
B11 = {1.present, 2.present, 3.double, 4.once, 5.end, 6.once, 7.start},
B12 = {1.present, 2.present, 3.longer, 4.zero, 5.not_there, 6.zero 7.not_there},
B13 = {1.present, 2.present, 3longer, 4.once, 5.start, 6.zero, 7.not_there},
B14 = {1.present, 2.present, 3.longer, 4.once, 5.start, 6.once, 7.end},
B15 = {1.present, 2.present, 3.longer, 4.once, 5.start, 6.once, 7.middle},
B16 = {1.present, 2.present, 3.longer, 4.once, 5.start, 6.more_than_once, 7.middle_end},
B17 = {1.present, 2.present, 3.longer, 4.once, 5.end, 6.zero, 7.not_there},
B18 = {1.present, 2.present, 3.longer, 4.once, 5.end, 6.once, 7.start},
B19 = {1.present, 2.present, 3.longer, 4.once, 5.end, 6.once, 7.middle},
B20 = {1.present, 2.present, 3.longer, 4.once, 5.end, 6.more_than_once, 7.start_middle}
B21 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.zero, 7.not_there}
B22 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.once, 7.start},
B23 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.once, 7.end},
B24 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.once, 7.mid},
B25 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.more_than_once, 7.start_middle},
B26 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.more_than_once, 7.start_end},
B27 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.more_than_once, 7.middle_end},
B28 = {1.present, 2.present, 3.longer, 4.once, 5.middle, 6.more_than_once, 7.start_middle_end}

2.3.7 Convert test frames into actual test cases

This is fairly straight-forward, and simply consists of devising inputs that satisfy the descriptions given in the test frames. The expected output (according to the specification) should also be worked out.

A few frames are converted to actual cases as examples.

1. *Better*:

   A1 = Parameters:

   $$\begin{aligned}u1 &= (\langle\rangle, \mathit{nulUR})\\ u2 &= (\langle\text{fred}\rangle, \mathit{UR}(\mathit{male}, \mathit{LikesPizza}))\\ u3 &= (\langle\text{jane}\rangle, \mathit{UR}(\mathit{female}, \mathit{DoesntLikePizza}))\end{aligned}$$

   Test:

   $\mathit{Better}(u1, u2, u3)$

   Valid result(s):

   False

   A9 = Parameters:

   $$\begin{aligned}u1 &= (\langle\text{john}\rangle, \mathit{UR}(\mathit{male}, \mathit{LikesPizza})\\ u2 &= (\langle\text{fred}\rangle, \mathit{UR}(\mathit{male}, \mathit{DoesntLikePizza}))\\ u3 &= (\langle\text{jane}\rangle, \mathit{UR}(\mathit{female}, \mathit{DoesntLikePizza}))\end{aligned}$$

   Test:

   $\mathit{Better}(u1, u2, u3)$

   Valid result(s):

   False

2. *BestInList*:

   B8 = Parameters:

   $$\begin{aligned}u &= \langle\text{fred}\rangle\\ \mathit{seq} &= \langle(\langle\text{fred}\rangle, \mathit{UR}(\mathit{male}, \mathit{DoesntLikePizza})),\\ &\qquad(\langle\text{john}\rangle, \mathit{UR}(\mathit{male}, \mathit{DoesntLikePizza}))\rangle\end{aligned}$$

   Test:

   $\mathit{BestInList}(u, \mathit{seq})$

   Valid result(s):

   $(\langle\rangle, \mathit{nulUR})$

   B15 = Parameters:

   $$\begin{aligned}u &= \langle\text{fred}\rangle\\ \mathit{seq} &= \langle(\langle\text{fred}\rangle, \mathit{UR}(\mathit{male}, \mathit{DoesntLikePizza})),\\ &\qquad(\langle\text{jane}\rangle, \mathit{UR}(\mathit{female}, \mathit{DoesntLikePizza})),\\ &\qquad(\langle\text{jasmine}\rangle, \mathit{UR}(\mathit{female}, \mathit{LikesPizza})),\\ &\qquad(\langle\text{john}\rangle, \mathit{UR}(\mathit{male}, \mathit{DoesntLikePizza}))\rangle\end{aligned}$$

Test:

$$BestInList(u, seq)$$

Valid result(s):

$$(\langle \text{jane} \rangle, UR(female, DoesntLikePizza))$$

B16 = Parameters:

$$\begin{aligned} u &= \langle \text{fred} \rangle \\ seq &= \langle (\langle \text{fred} \rangle, UR(male, DoesntLikePizza)), \\ &\quad (\langle \text{jane} \rangle, UR(female, DoesntLikePizza)), \\ &\quad (\langle \text{jasmine} \rangle, UR(female, LikesPizza)), \\ &\quad (\langle \text{jim} \rangle, UR(male, DoesntLikePizza)), \\ &\quad (\langle \text{joanne} \rangle, UR(female, DoesntLikePizza)) \rangle \end{aligned}$$

Test:

$$BestInList(u, seq)$$

Valid result(s):

$$\begin{aligned} &(\langle \text{joanne} \rangle, UR(female, DoesntLikePizza)) \\ &(\langle \text{jane} \rangle, UR(female, DoesntLikePizza)) \end{aligned}$$

This example illustrates a case where more than a single result might be acceptable. According to the definition of *BestInList* then only the first result, ⟨joanne⟩ is acceptable. But, according to the original definition of *bestdate*, the second, ⟨jane⟩, would also be correct.

2.3.8 Apply the test cases and evaluate the results

The system described by the specification was actually implemented (in the functional language Hope$^+$C) together with various supplementary functions to construct suitable data-structures. Also, code was written to execute the code repeatedly with different test cases as input.

Rather than examine what would happen if "correct" code were tested, we consider faulty code here. Suppose the code for *Better* satisfied the following specification, instead of the one supplied earlier:

$$\begin{array}{|l} Better : RecPair \times RecPair \times RecPair \to \text{BOOLEAN} \\ \hline \forall\, r_1, r_2, r_3 : RecPair \\ \quad Better(r_1, r_2, r_3) \Leftrightarrow \\ \quad\quad ((compatibility((r_1).userrecord, (r_3).userrecord) > \\ \quad\quad\quad\quad compatibility((r_2).userrecord, (r_3).userrecord) \lor \\ \quad\quad\quad nulUR \in \{(r_i).userrecord \mid i = 1, 2, 3\}) \land \\ \quad\quad (etc.\ \text{as before}) \ldots \end{array}$$

The only changes made, are that an $\land$ has been replaced by an $\lor$, and a $\notin$ by a $\in$. The result is that *Better* would evaluate to *true* if *any* of its arguments were *nulUR*.

Then, invalid output would result for many of the tests. For instance:

A1 would have result *True* not *False*.

B8 would give the result

$$(\langle \text{john} \rangle, UR(male, DoesntLikePizza))$$

2.4 Conclusion

This chapter demonstrated that functional tests for a system can be systematically derived from its formal specification. It also presented an example of a formal specification for (part of) a large system.

The formal specification means that all of the early parts of the testing procedure are easy to carry out. The functions have been identified, with their parameters (and the environment conditions, if they are regarded as different) so the first stages have already been done. The formal specification means that the valid results for each test case can be worked out with certainty.

The use of the TSL and an automatic tool means that the test specification and test frames are written in a standard format, which could easily be modified to suit the format required for a particular standard.

Constructing the test specification informs the design specification process. Many special cases and exception conditions are uncovered by the process of writing tests for them. Similarly, using the design specification to work out expected test results will frequently uncover limitations and errors in the design.

It is fairly easy to modify the test specification as the design specification evolves, so it is practical for test specification to begin at the same time as design specification.

On the other hand, it is difficult to find satisfactory formalisms describing the structures identified in the testing process. For instance, what, precisely, is a *category*?

This chapter concentrated on just one of the functions of a complete dating agency system. The complete system would have many such functions, interacting in a complex way. The category-partition method gives no guidance as to how to integrate the testing of individual functions into tests for the system as a whole.

Using the category-partition method, the test cases are given in the form of parameters for the functions. No guidance is given on how to generate the required values for the parameters when the function is embedded within a large system; it is assumed that a test harness of some sort can be used to generate arbitrary data structures (in the case of the example, the data structures are sequences of *USERRECORDS*).

Also the choice partitions that are made rely heavily on the experience of the tester for their effectiveness. An arbitrary partition of a category into choices will not necessarily be much use for revealing likely faults. For instance, in the extreme, it would be legitimate to just have one choice for each category (corresponding to "any value"), and so there would be only a single test case for the function.

Chapter 3

Fundamentals.

3.1 Goodenough and Gerhart's theoretical framework.

The earliest theoretical discussion of testing, and how it relates to correctness, was by Goodenough & Gerhart [17]. The theory was not intended for use as a direct means of test selection, and was subsequently shown to have a number of limitations, as discussed later in this chapter.

3.1.1 Initial definitions.

These definitions are those of Goodenough & Gerhart [17].

S is a specification and I is an implementation of it. Assume that they can be regarded as functions:

$$S : D \to R, \qquad I : D \to R.$$

D is the domain, or set of possible inputs, R is the range.

Definition 3.1.1 $\textsc{out}(d, r) =$ true iff $I(d) = r$ is an acceptable output, *i.e.* $S(d) = r$.

Equivalently,

$$\textsc{out}(d, r) \Leftrightarrow [S(d) = r].$$

◇

Definition 3.1.2 $\textsc{ok}(d) \equiv \textsc{out}(d, I(d))$. ◇

Definition 3.1.3 A test set $T \subseteq D$ is *ideal* if

$$\forall\, t \in T \cdot \textsc{ok}(t) \Rightarrow \forall\, d \in D \cdot \textsc{ok}(d).$$

◇

Definition 3.1.4 C is a *criterion* for test data selection. It takes the form of a predicate, which can be applied to test sets, $T \subseteq D$. ◇

Definition 3.1.5 $\textsc{complete}(T, C)$ iff the elements of T collectively satisfy the criterion C. ◇

Clearly, if $T = D$, T is an ideal test set. This is not very useful though. The idea behind C is to choose it such that $T(\subseteq D)$ is ideal iff $\textsc{complete}(T, C)$. The next section develops this.

Definition 3.1.6

$$\textsc{successful}(T) \equiv [\forall\, t \in T \bullet \textsc{ok}(t)].$$

◇

The definition of $\textsc{successful}(T)$ is implicitly dependent on the implementation, I.

Definition 3.1.7

$$\begin{aligned}
&\textsc{reliable}(C) \equiv \forall\, T_1, T_2 \subseteq D \bullet \\
&\qquad \textsc{complete}(T_1, C) \wedge \textsc{complete}(T_2, C) \Rightarrow \\
&\qquad\quad (\textsc{successful}(T_1) \equiv \textsc{successful}(T_2)),
\end{aligned}$$

i.e. a criterion C is reliable iff any test set satisfying it will either always test successfully or always test unsuccessfully. ◇

Definition 3.1.8

$$\begin{aligned}
&\textsc{valid}(C) \equiv \forall\, d \in D \bullet \\
&\qquad \neg\ \textsc{ok}(d) \Rightarrow \\
&\qquad\quad \exists\, T \subseteq D \bullet (\textsc{complete}(T, C) \wedge \neg\ \textsc{successful}(T)),
\end{aligned}$$

i.e. a criterion is valid iff there is a test set satisfying it that will reveal any particular fault. ◇

Both $\textsc{reliable}(C)$ and $\textsc{valid}(C)$ involve $\textsc{successful}(T)$, and so are dependent on the particular implementation under consideration.

Theorem 3.1.9 (The Fundamental Theorem of Testing) [17] Given definitions 3.1.1 to 3.1.8,

$$\begin{aligned}
&\exists\, T \subseteq D;\ \exists\, C \bullet \\
&\quad \textsc{complete}(T, C) \wedge \textsc{reliable}(C) \wedge \textsc{valid}(C) \wedge \textsc{successful}(T) \Rightarrow \\
&\qquad \forall\, d \in D \bullet \textsc{ok}(d).
\end{aligned}$$

This simply means that if a test set satisfies a criterion that is valid and reliable, and is executed successfully, then the program will execute successfully for any input.

Proof Assume there exists some $d \in D$ for which I fails (*i.e.* $\neg\ \textsc{ok}(d)$). Then $\textsc{valid}(C)$ implies that there exists a complete set of test data, T, that is not successful. $\textsc{reliable}(C)$ implies that if one complete test set fails then all will fail. But this contradicts the theorem's premise, that there is a complete test set which is successfully executed. □

3.2 Weyuker and Ostrand's Revealing Sub-domains.

There are a number of limitations to the Goodenough and Gerhart's theory.

Firstly, there is the difficulty of choosing the criteria and proving that they are reliable and valid, and then proving that a particular test set is complete with respect to the criterion. In general, this is just as difficult as a proof.

The second major drawback is the dependence of the definitions of SUCCESSFUL and VALID and RELIABLE on the implementation under consideration, so that the proofs of these properties is unique for each particular version of the implementation, no matter how small the differences. In particular, it is necessary to know what faults are in the implementation (or at least the *types* of fault) to prove that a given criterion is valid and reliable. Further, if any change is made to the implementation (*e.g.* to correct a fault that has been revealed), then the criterion used will not necessarily still be valid and reliable. Worse still, some criteria that were not valid and reliable before the change may become valid and reliable after it.

These problems were identified by Weyuker and Ostrand [59], who developed the theory further to try to overcome them, specifically by abandoning the goal of strict correctness, and aiming at *correctness with respect to likely faults* instead.

There are several steps to go through first.

Definition 3.2.1 Let $B \subseteq D$. Then test criterion C is *revealing* for B if, whenever B contains inputs that will lead to failures, every test set complete with respect to C is unsuccessful. *i.e.*

$$\begin{aligned}\textsc{revealing}(C, B) \equiv [\exists\, d \in B \bullet \neg\, \textsc{ok}(d)] \Rightarrow \\ [\forall\, T \subseteq B \bullet (\textsc{complete}(C, T) \Rightarrow \neg\, \textsc{successful}(T))]\end{aligned}$$

◇

Notice that if $B = D$, then the criterion chosen is valid and reliable for the implementation.

In general, it will be just as hard to show that a criterion is revealing as it is to show that it is valid and reliable, and for the same reasons.

However, suppose that the criterion is true for any test set. Then, the definition becomes

$$\begin{aligned}\textsc{revealing}(B) \equiv [\exists\, d \in B \bullet \neg\, \textsc{ok}(d)] \Rightarrow \\ [\forall\, T \subseteq B \bullet \neg\, \textsc{successful}(T)].\end{aligned}$$

Under this definition, a sub-domain B is revealing if a fault affecting any member of B will also affect all the other members of B. Proving that a sub-domain is revealing means proving that **every** fault that affects any element of B, affects *every* element of B, or, alternatively, that *no* fault affects *any* element of B. This is still generally difficult.

Now define an even more restricted version of revealing.

Definition 3.2.2 A sub-domain B is revealing for a *fault* F in I, if, if F affects any member of B, it affects all the members of B. ◇

So, if $d \in B$, and $\textsc{ok}(d)$, then the fault F is guaranteed not to affect any of the sub-domain.

This is a definition of some pragmatic use. By enumerating the *likely* faults, they can be guaranteed not to affect a particular implementation if successful tests are run on *carefully* chosen revealing sub-domains.

However, the vague notions of *likely* and *carefully* need to be formalised in some way.

3.3 Fault-based testing.

Morell [38] discusses the problem at some length, and draws on the work of Howden [25], DeMillo *et al.* [6] and others working on mutation testing. Morell's *theory of fault-based testing* is a generalisation of some of the ideas of mutation testing. It is based on a notation for describing a set of alternatives to a particular implementation, and on the use of symbolic execution to determine revealing sub-domains for them.

3.3.1 Notation and definitions.

These definitions are due to Morell [38]. They are based on a specification S, and an implementation I. As usual, they are regarded as functions, $S, I : D \to R$.

Definition 3.3.1 A *location* in I denotes some expression of I. An expression corresponds to a (group of) symbol(s) from the syntax of the language that I is written in. ◇

Definition 3.3.2 An *alternate expression* or *alternative* is an expression, f, that can be substituted for an expression, l in I, without violating the syntactic correctness of I. ◇

Definition 3.3.3 The result of making such a substitution in I, is the *alternate implementation*, I^l_f. I is the *original* specification. ◇

Definition 3.3.4 Any location in I can have a set of alternatives, called the *alternative set.* This set generally contains the original. ◇

Definition 3.3.5 A *fault-based arena* is a 5-tuple: $E =< I, S, D, L, A >$, where I, S and D are the implementation, specification and input domain respectively, and: $L = (l_1, l_2, \ldots, l_n)$ is an n-tuple of locations in I, and $A = (A_1, A_2, \ldots, A_n)$ is an n-tuple, each A_i being an alternative set for l_i.

There is no restriction on the number of alternatives allowed at each location. ◇

Mutation testing is fault-based testing in which the alternative sets are finite.

Definition 3.3.6 I_E is the set of all programs that can be generated from I using a fault-based arena, E, by substituting one or more alternatives at their respective locations in I. ◇

Test data sets are sought that *distinguish* or *differentiate* the original program from its alternates.

Definition 3.3.7 For implementation I, and $x \in \text{dom}(I)$, x *distinguishes* I from an implementation J iff $I(x) \neq J(x)$.

$T \subseteq \text{dom}(I)$, T distinguishes I from J iff $\exists\, x \in T$ such that x distinguishes I from J.

Similarly, a test set T can distinguish I from a set of programs $\mathcal{J}$. ◇

3.3.2 Fault-based testing and correctness.

Definition 3.3.8 A fault-based arena, $E =< I, S, D, L, A >$ is *alternate sufficient* iff $\exists\, J \in I_E$ such that J is correct with respect to S. ◇

Definition 3.3.9 Alternatives a_1 and a_2 are *coupled* if they can be individually distinguished from the correct implementation by a single test, but their combination cannot be precluded by the same test. *i.e.*

$$\exists\, t \in D \bullet I^{l_1}_{a_1}(t) \neq S(t) \wedge I^{l_2}_{a_2}(t) \neq S(t) \wedge I^{l_1,l_2}_{a_1,a_2}(t) = S(t).$$

◇

Morell argues that there are two conditions governing whether fault-based testing ensures correctness:

1. The fault-based arena must be alternate-sufficient.
2. Coupling does not occur anywhere in the test set.

Unfortunately, as Morell shows, it is undecidable whether an arbitrary fault-based arena is alternate-sufficient, or involves coupling.

Therefore, alternate-sufficiency is *assumed* to hold, until evidence that it does not is found. Then some new alternatives have to be found, extending the fault-based arena, and the assumption is taken up again. When it is assumed to be true for a particular implementation, alternate-sufficiency is known as the *competent programmer hypothesis*.

However, the absence of coupling can sometimes be inferred from the implementation structure: alternatives cannot be coupled if they lie on different paths, or if they affect different parts of the data structure.

Definition 3.3.10 $E =< I, S, D, L, A >$ is *finitely distinguishable*, or *finite*, iff there is a finite test set T, that distinguishes I from I_E. ◇

Every arena in which all the alternative sets are finite, is finitely distinguishable (which includes mutation testing). There are also arenas with infinite alternative sets that are finitely bounded.

3.3.3 Symbolic testing.

But this still leaves the arenas that aren't finitely distinguishable. Morell extends the theory by allowing symbolic arenas, where symbolic alternatives take the place of a possibly infinite number of real alternatives.

There are two aspects to symbolic testing:

- *Symbolic inputs* are used to model inputs that would follow a given path through the implementation.
- *Symbolic alternatives* are used to model whole classes of ordinary alternatives.

  i.e. given a location l_i from an arena, the whole alternative set, A_i, can be represented by a single symbolic alternative.

By using a development of standard symbolic execution techniques, the implementation can be executed with the symbolic inputs, and symbolic alternatives to produce symbolic outputs.

For symbolic alternative $\mathcal{A}$ and symbolic input x, two expressions need to be generated:

- $\mathcal{O}_x$, the symbolic output generated, when $\mathcal{A}$ is not used,

  i.e. $\mathcal{O}_x = I(x)$;
- $\mathcal{O}_x^{\mathcal{A}}$, the symbolic output generated, when $\mathcal{A}$ is used,

  i.e. $\mathcal{O}_x^{\mathcal{A}} = I_{\mathcal{A}}^{l_i}(x)$, where l_i is the location of $\mathcal{A}$.

Consider $\mathcal{O}_x = \mathcal{O}_x^{\mathcal{A}}$ and $\mathcal{O}_x \neq \mathcal{O}_x^{\mathcal{A}}$ (called the *propagation equation* and *assertion* respectively). They can be solved for $\mathcal{A}$ (in terms of the input).

Solutions to the propagation equation correspond to faults that are indistinguishable by the symbolic input.

Solutions to the propagation assertion correspond to faults that are distinguishable by the symbolic input. In fact, real inputs corresponding to the symbolic input form revealing sub-domains for these faults.

3.3.4 Limitations of fault-based testing.

Morell's work is a considerable step forward from Weyuker & Ostrand's, as it describes a way to actually generate revealing sub-domains. However, it is oriented towards the implementation rather than the specification: the faults are defined in terms of alternate expressions in specific locations of the implementation. They are therefore dependent on the structure (and hence the syntax) of the implementation, which is not entirely satisfactory. This is clearly shown by the need for the competent programmer hypothesis.

One reason for this is that the arenas are based wholly around the syntactic representation of the implementation, rather than on semantic representations of the implementation *and the specification.* A more detailed look at the meanings of the word *fault* is required.

3.4 Faults.

Recall from definition 1.1.2 that, given a specification $S : D \to R$ and an implementation of it, $I : D \to R$, then a *failure* is a maplet $(i \mapsto o) \in (I \setminus S)$, *i.e.* a maplet belonging to I but not to S.

Given this definition, the simplest formalisation of the concept *fault*, is as sets of failures. Thus a fault is a partial function consisting of the failures it causes. So S could be implemented with fault $\mathcal{F} = \{i_1 \mapsto o_1, i_2 \mapsto o_2, \ldots, i_n \mapsto o_n\}$, in which case the implementation, I, would be $S \oplus \mathcal{F}$.

However, it does not take into account the fact that distinct faults may cause exactly the same failures. For instance, many different faults could lead to the failures of the system crashing on every input, *i.e.* $I = S \oplus (\lambda\, d : D \bullet \perp)$.

To describe this, a mapping between actual faults and the sets of failures could be used, *i.e*:

- $\Delta == \{d : D;\ r : R \mid S(d) \neq r \bullet d \mapsto r\}$, the set of all possible failures. (See Spivey, [55] for the set notation.)

- Let *Faults* denote the set of possible faults that can occur in converting S to I.
- $\mathit{FaultMap}(S) : \mathit{Faults} \rightarrow \mathbb{P}(\Delta)$, describes S's fault mapping.

Under this scheme of things, $\mathrm{dom}(\mathit{FaultMap}(S)) = \mathit{Faults}$ is the set of all possible faults that might occur in converting S into I. Also, given fault $\mathcal{F}$, $\mathrm{dom}(\mathcal{F})$ is a revealing sub-domain.

To guarantee correctness a property analogous to alternate sufficiency is needed on *Faults*, *i.e.* it has to genuinely describe all the possible faults.

This is all very well, but *Faults* still remains undefined. What is required is a framework in which S and I can be directly compared, in which case the differences will be faults. Such a framework will be capable of modelling both S and I in the same terms.

One such model might be Finite-State machines.

3.4.1 Finite-state-machine models of S and I.

Definition 3.4.1 This definition is largely from Chow [4].

A deterministic *finite-state-machine* (FSM), $\mathcal{M}$ is a quintuple:

$$\mathcal{M} = (Y, Z, Q, F, O),$$

where:

- Y is the input set.
- Z is the output set.
- Q is the set of states, including q_0, the initial state.
- $F : Y \times Q \rightarrow Q$ is the transition function (or next-state function).
- $O : Y \times Q \rightarrow Z$ is the output function.

The *general behaviour* of the machine can be described by a function,

$$\|\mathcal{M}\| : Q \rightarrow (Y^+ \rightarrow Z^+).$$

The *behaviour* of the machine can be described by a function,

$$|\mathcal{M}| = \|\mathcal{M}\|\,(q_0).$$

Finite-state-machines are normally defined by drawing a state transition diagram with labelled arcs, or using a state transition table. ◇

When input and output are regarded as sequences, input is processed "head-first", output is generated "head-last."

Notation 3.4.2 Given finite-state-machine $\mathcal{M}$ with states $q_1, q_2 \in Q$, and with input $y \in Y$ such that $F(q_1, y) = q_2$, and $O(q_1, y) = z$, this can be written in the following way:

$$q_1 \xrightarrow{y/z} q_2,$$

or, if the output is not important, as

$$q_1 \xrightarrow{y} q_2.$$

◇

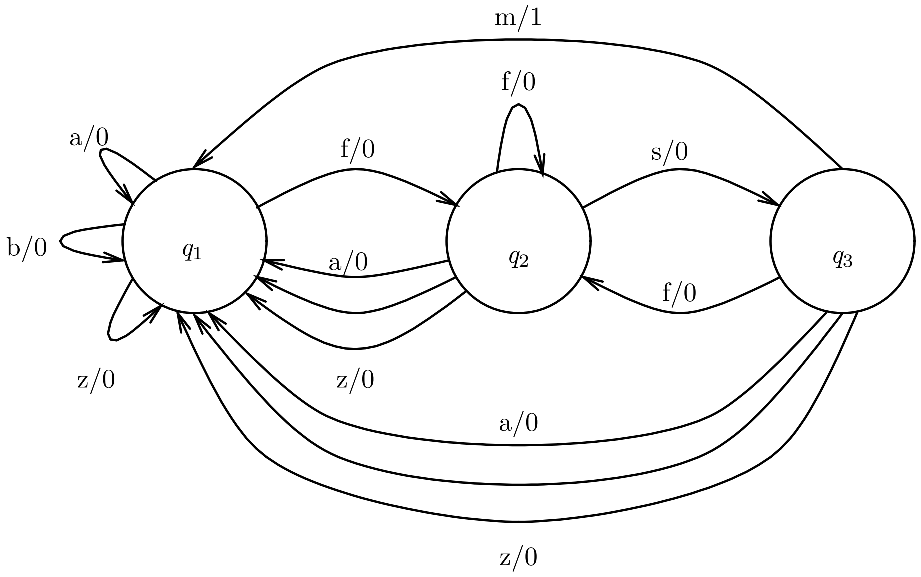


Figure 3.1: State transition diagram of a finite-state-machine to recognise the sequence “fsm”.

In terms of the specification $S : D \to R$ and implementation $I : D \to R$, the notion of finite-state-machines is useful so long as there are suitable Y and Z such that $Y^+ = D$ and $Z^+ = R$.

The model is fairly simple, so given $\mathcal{S}$, the finite-state-machine model of S there are not many different ways that faults can appear in $\mathcal{I}$ (the finite-state-machine model of I):

1. $\mathcal{I}$ could have missing states;
2. $\mathcal{I}$ could have extra states;
3. $\mathcal{I}$ could have a faulty transition function (*i.e.* a misdirected or missing arc between states).
4. $\mathcal{I}$ could have a faulty output function.

Example 3.4.3 Figure 3.1 shows a finite-state-machine as a state transition diagram, and table 3.1 shows the corresponding state transition table. The machine accepts a character stream as input, so $Y = \{a, \ldots, z\}$, and $D = Y^+$, and outputs a binary stream, with a 1 for every occurrence of the sequence “fsm” in the input. So $Z = \{1, 0\}$, and $R = Z^+$.

The simplest faults involve only single maplets of the transition function or of the output function. For instance, $F(m, q_3) = q_3$ is a fault, and $O(m, q_3) = 0$ is a fault.

The situation is more complicated if missing or extra states are involved. For instance, if a state is missing, then all the maplets from F that involve it must also change, and there are many different ways in which they could change. $\diamond$

The theory of finite-state-machines has been thoroughly investigated, and their use for generating test cases is well established [3, 4, 12, 54].

3.5 Test generation from finite-state-machine models of *S*.

In this section I describe Chow’s W-method of test set generation [4]. It will detect all transition errors and output errors, plus any missing states, and any extra states up to a finite limit, (say up to e extra states).

Y	Q		
	q_1	q_2	q_3
a	$q_1/0$	$q_1/0$	$q_1/0$
b	$q_1/0$	$q_1/0$	$q_1/0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
e	$q_1/0$	$q_1/0$	$q_1/0$
f	$q_2/0$	$q_2/0$	$q_2/0$
g	$q_1/0$	$q_1/0$	$q_1/0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
l	$q_1/0$	$q_1/0$	$q_1/0$
m	$q_1/0$	$q_1/0$	$q_1/1$
n	$q_1/0$	$q_1/0$	$q_1/0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
r	$q_1/0$	$q_1/0$	$q_1/0$
s	$q_1/0$	$q_3/0$	$q_1/0$
t	$q_1/0$	$q_1/0$	$q_1/0$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
z	$q_1/0$	$q_1/0$	$q_1/0$

Table 3.1: State transition table of a finite-state-machine to recognise the sequence “fsm”.

3.5.1 Preliminary definitions.

In the definitions that follow (which are from Chow), $\mathcal{M}$ is a general finite-state-machine, $\mathcal{S}$ a finite-state-machine modelling S, the specification, and $\mathcal{I}$ is a finite-state-machine modelling I, an implementation of it.

$$\begin{aligned}\mathcal{M} &= (Y, Z, Q, F, O)\\ \mathcal{S} &= (Y, Z, Q_\mathcal{S}, F_\mathcal{S}, O_\mathcal{S})\\ \mathcal{I} &= (Y, Z, Q_\mathcal{I}, F_\mathcal{I}, O_\mathcal{I})\end{aligned}$$

Definition 3.5.1 $\mathcal{M}$ is *completely specified* if there is a transition defined for every input symbol $y \in Y$ for every state. ◇

Definition 3.5.2 $\mathcal{M}$ is *strongly connected* if there are input sequences that take the machine from each state to any other state. ◇

Definition 3.5.3 Given sets of input sequences, V_1 and V_2, their *concatenation* is the set $V_1 \cdot V_2 == \{v_1 \frown v_2 \mid v_1 \in V_1, v_2 \in V_2\}$. ◇

Notation 3.5.4 For input sequence V, V^n denotes the n-times concatenation of V. ◇

Notation 3.5.5 $V[k] == (\{\varepsilon\} \cup V \cup V^2 \cup \cdots \cup V^k)$, where ε is the empty sequence. ◇

Definition 3.5.6 Given a set V of input sequences, states $q^\mathcal{S}$ and $q^\mathcal{I}$, from $\mathcal{S}$ and $\mathcal{I}$ are V-*equivalent* if $\mathcal{S}$ in state $q^\mathcal{S}$ and $\mathcal{I}$ in state $q^\mathcal{I}$ respond with identical output sequences to each input sequence in V. ◇

Definition 3.5.7 Two states are *equivalent* if they are V-equivalent for any set V. ◇

Definition 3.5.8 Machines $\mathcal{S}$ and $\mathcal{I}$ are equivalent if their initial states, $q_0^\mathcal{S}$ and $q_0^\mathcal{I}$ are equivalent. ◇

Definition 3.5.9 $\mathcal{M}$ is *minimal* if it has fewer or equal states to any other machine $\mathcal{M}'$ which is equivalent to $\mathcal{M}$. ◇

Definition 3.5.10 V, a set of input sequences, is a *state cover set* of $\mathcal{M}$ if for each state $q_i^{\mathcal{M}}$ of $\mathcal{M}$, there is an input sequence $v_i \in V$ such that $q_0^{\mathcal{M}} \xrightarrow{v_i} q_i^{\mathcal{M}}$. $\diamond$

Definition 3.5.11 V, a set of input sequences, is a *transition cover* set of $\mathcal{M}$ if, for each transition $q_i \xrightarrow{y/z} q_j$, there are sequences v and $v \frown \langle y \rangle$ in V such that $q_0 \xrightarrow{v} q_i$ and $q_0 \xrightarrow{v \frown \langle y \rangle} v_j$.
A transition cover set contains a state cover set. $\diamond$

Definition 3.5.12 W, a set of input sequences, is a *characterisation set* of $\mathcal{M}$ if, for each pair of different states, q_i, q_j, the output sequences produced by W when applied in states q_i and q_j are different. $\diamond$

3.5.2 Test set description.

Chow's test set, T, is built up from several parts, and is entirely based on $\mathcal{S}$:

- C a transition cover set for $\mathcal{S}$;
- P a *prefix set* for $\mathcal{S}$, where e is the maximum number of extra states in $\mathcal{I}$ compared to $\mathcal{S}$,

$$\begin{aligned} P &= (\{\varepsilon\} \cup Y \cup Y^2 \cup \cdots \cup Y^e) \\ &= Y[e] \end{aligned}$$

- W a characterisation set for $\mathcal{S}$;

$$T = C \cdot P \cdot W$$

The idea is that the transition cover (C) ensures that all the transitions of $\mathcal{S}$ are present in $\mathcal{I}$, and the remainder $(P \cdot W)$ ensures that $\mathcal{I}$ is in the same state as $\mathcal{S}$ would be after the each transition is used.

P is needed to detect extra states. If there are up to e extra states, and they are reachable from $q_0^{\mathcal{I}}$, then they must be reachable by some input sequence of up to length e from one of the existing states. Notice that if $e = 0$ (*i.e.* no extra states) then $P = \varnothing$.

3.5.3 Correctness.

In order to guarantee correctness, the following conditions have to hold:

- The specification, $\mathcal{S}$ must be *minimal*, which is a necessary and sufficient condition to guarantee the existence of a characterisation set W [3, chapter 2].
- $\mathcal{S}$ must have at least two states.
- $\mathcal{S}$ and $\mathcal{I}$ must be completely specified, and deterministic.
- $\mathcal{S}$ and $\mathcal{I}$ must both be strongly connected.
- $\mathcal{S}$ must have a reset operation (*i.e.* an extra input that causes the machine to change back to $q_0^{\mathcal{S}}$ from every state; in the worst case this corresponds to re-starting the system), which must be correctly implemented in $\mathcal{I}$.
- The machines must have the same input set.
- The number of states in $\mathcal{I}$ must be bounded by $m \geq n$, where n is the number of states in $\mathcal{S}$. So $e = m - n$.

Chow proves that correctness is guaranteed under these circumstances [4].

If $\mathcal{I}$ is not minimal, but otherwise correct w.r.t. $\mathcal{S}$, the test set, T, will not detect it.

3.5.4 Practical application of the method.

The three sets C, P and W can be generated automatically from $\mathcal{S}$, as described by Chow [4], Fujiwara *et al.* [12] and Bhattacharyya [3].

Example 3.5.13 The test set for the finite-state-machine in example 3.4.3 can be constructed as follows (assuming that there are no more than 2 extra states in the implementation :

$$\mathcal{Y} = \{\langle a\rangle, \ldots, \langle z\rangle\}$$

$$C = \mathcal{Y} \cup \{\langle f\rangle\} \cdot \mathcal{Y} \cup \{\langle fs\rangle\} \cdot \mathcal{Y}$$

$$P = \{a, \ldots z\}[2]$$

$$W = \{\langle sm\rangle, \langle m\rangle\}.$$

To see that W is a characterisation set, consider the behaviour function in each case

$$\begin{aligned} \|\mathcal{M}\|\,(q_1)\langle sm\rangle &= \langle 00\rangle \\ \|\mathcal{M}\|\,(q_1)\langle m\rangle &= \langle 0\rangle \\[1em] \|\mathcal{M}\|\,(q_2)\langle sm\rangle &= \langle 01\rangle \\ \|\mathcal{M}\|\,(q_2)\langle m\rangle &= \langle 0\rangle \\[1em] \|\mathcal{M}\|\,(q_3)\langle sm\rangle &= \langle 00\rangle \\ \|\mathcal{M}\|\,(q_3)\langle m\rangle &= \langle 1\rangle. \end{aligned}$$

Altogether there are $\text{size}(C) \times \text{size}(P) \times \text{size}(W) = 78 \times 702 \times 2 = 109512$ test cases, although many of these will be duplicates of one another. ◇

According to Chow, the upper bound on the number of test cases is

$$[\text{size}(Q_{\mathcal{S}})]^2 \times [\text{size}(Y)]^{(e+1)},$$

where e is the maximum number of extra states in $\mathcal{I}$ compared to $\mathcal{S}$. (In the case of the example, this gives a value of 158184.)

3.6 Limitations of finite-state-machine based testing.

The finite-state-machine model enables revealing test sets to be generated in a straight forward manner. But the model is rather simple and has several drawbacks.

Firstly, there is no way to model data structures independently of the control structure. This makes modelling large systems very difficult. In fact, it is difficult to model *any* non-trivial data structure using finite-state-machines.

Secondly for a finite-state-machine to be complete, there has to be one transition defined for every different input in every state. This leads to a very large number of test cases, many of which are very similar. Fujiwara *et al.* [12] describe various ways to reduce the size of the test set, but the reductions just avoid unnecessary repetition, and other minor aspects of minimisation.

Both problems could be overcome with a model that separates the data state from the control structure. Input (and output) should be treated as part of the data structure, removing the need for a lot of the "almost-duplication" of transitions seen in the example. I investigate the possibilities of such a model in the next few chapters.

Chapter 4

Specification using X-machines.

4.1 Introduction to X-machines.

X-machines were first proposed by Samuel Eilenberg [9, chapter X] as a general machine model of computation. Sadly, Eilenberg never fully described their potential, or even all of his reasons for introducing them. However, they offer a number of attractions as an abstract model for use in specification, as pointed out by Mike Holcombe in [24]. X-machines allow the control and data manipulation aspects of a system to be succinctly and separately modelled. These ideas are developed here.

4.1.1 Basic concepts

Definition 4.1.1 This style of definition is due to Holcombe [24].

An X-machine, $\mathcal{M}$, is a 10-tuple, as follows:

$$\mathcal{M} = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T)$$

1. X is the fundamental data set that the machine operates on.
2. Y and Z are the input and output sets, respectively.
3. α and β are the input and output relations respectively, used to convert the input and output sets into, and from, the fundamental set. *i.e.*

   $$\alpha : Y \leftrightarrow X \qquad \beta : X \leftrightarrow Z$$

4. Q is the (finite) set of states.
5. Φ is the *type* of $\mathcal{M}$, a set of relations on X.
   i.e.

   $$\Phi : \mathbb{P}(X \leftrightarrow X)$$

   Φ can also be viewed as an abstract alphabet. It is not necessary for all of the relations in Φ to be used in $\mathcal{M}$. In fact, Φ can be infinite, although only a finite number of the members will actually be used in any particular machine.
6. F is the ‘next state’ function.

   $$F : Q \rightarrow (\Phi \rightarrow \mathbb{P}\, Q)$$

It is described as a Curry-ed function. So, for state $q \in Q$,

$$F(q) : \Phi \rightarrow \mathbb{P}\, Q$$

However, when it is convenient, F can be treated like a function with two arguments; *i.e.* $F(q, \phi) = (F(q))(\phi)$.

F is often described by means of a diagram.

7. I and T are the sets of initial and terminal states respectively.

$$I \subseteq Q,\ T \subseteq Q$$

◇

The remaining definitions in this chapter are my own.

Definition 4.1.2 If $q_a, q_b \in Q$, $\phi \in \Phi$ and $q_b \in F(q_a, \phi)$, write $q_a \xrightarrow{\phi} q_b$. ϕ is the *arc*, from q_a to q_b. ◇

Definition 4.1.3 If $q_a, q_b \in Q$ are such that there exist $q_1, \ldots q_n \in Q$ and $\phi_1, \ldots \phi_{n+1} \in \Phi$ with

$$q_a \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} \ldots q_n \xrightarrow{\phi_{n+1}} q_b,$$

then $p = (\langle q_a, q_1, \ldots, q_n, q_b\rangle, \langle \phi_1, \ldots, \phi_{n+1}\rangle)$ is the *path* from q_a to q_b.

There are *three* distinct elements to p:

1. the sequence of states passed through, $\langle q_a, q_1, \ldots, q_n, q_b\rangle \in Q^*$, called the *$Q$-path* of p, written p_Q;
2. the sequence of transitions used, $\langle \phi_1, \ldots, \phi_{n+1}\rangle \in \Phi^*$, called the *$\Phi$-path* of p, written p_Φ;
3. the relation due to the composition of p_Φ, $\phi_1\phi_2 \ldots \phi_{n+1} \in X \leftrightarrow X$, called the *label* of p, written $|p|$.

Notice that $\mathrm{len}(p_Q) = \mathrm{len}(p_\Phi) + 1$.

Write $q_a \xrightarrow{\phi_1\phi_2\ldots\phi_{n+1}} q_b$ or $q_a \xrightarrow{p} q_b$.

This is an extension to Eilenberg's path definition. ◇

NB The connection between paths and their labels is close, but not unique; distinct paths (*i.e.* passing through different sequences of states) can have exactly the same label, although not *vice versa.* However, in most cases the context provides enough information to make the meaning of a phrase like "using *path* $\phi_1 \ldots \phi_n$" clear without having to specify which states are passed through.

Definition 4.1.4 Given a path, p, from $q_a \in Q$ to $q_b \in Q$, then p is a *full-path* if $q_a \in I$ and $q_b \in T$. Otherwise, p is a *partial-path.* ◇

Definition 4.1.5 An X-machine, $\mathcal{M}$ is *deterministic* iff:

1. α is a function, not a relation:

$$\alpha : Y \rightarrow X$$

2. β is a function, not a relation:

$$\beta : X \rightarrow Z$$

3. Φ contains only partial functions on X rather than relations:

$$\Phi : \mathbb{P}(X \nrightarrow X)$$

4. F maps each pair $(q, \phi) \in Q \times \Phi$ onto at most a single next state:

   $F : Q \to (\Phi \nrightarrow Q)$

   A partial function is used because every $\phi \in \Phi$ will not necessarily be defined as the label to a path in every state.

5. I contains only a single element, $I = \{q_0\}$.

$\diamond$

4.1.2 Operation of X-machines

Given $y \in Y$, the *operation* of the X-machine $\mathcal{M}$ on y consists of:

1. Taking a path, p, from a start state, $q_i (\in I)$, to a finish state, $q_t (\in T)$, *i.e.* $q_i \xrightarrow{|p|} q_t$.
2. Apply α to the input to convert it to the internal type X.
3. Apply $|p|$, if it is defined for $\alpha(y)$. Otherwise, go back to step 1.
4. Apply β, to get the output.

Assuming the machine is deterministic, then the operation can be summarised as:

$\beta(|p| (\alpha(y)))$

Alternatively, the operation can be described state-transition by state-transition, as follows (assuming once again, that the machine is deterministic):

1. Apply α to y to obtain $x_0 = \alpha(y)$.
2. Consider $F(q_n)$. Take one of the transition functions, $\phi \in \text{dom}(F(q_n))$ for which $\phi(x_n)$ is defined.
3. Apply the ϕ to x_n to obtain $x_{n+1} = \phi(x_n)$, and apply F to get $q_{n+1} = F(q_n, \phi)$.
4. Go back to step 2, unless $q_{n+1} \in T$ and there is no $\phi \in \text{dom}(F(q_{n+1}))$ such that $\phi(x_{n+1})$ is defined.
5. Apply β to x_{n+1} to obtain the output, $z = \beta(x_{n+1})$.

This view of X-machine operation is particularly useful when modelling interactive systems, where the final output is of no special relevance compared to the state-transition by state-transition behaviour. In fact, in this sort of situation, it can be appropriate to consider β as an *output filter* that can be applied to the current x_n at any time. For instance, X might be a complicated data type describing the system data-state, and β the function that extracts the screen appearance from X.

Definition 4.1.6 An X-machine, $\mathcal{M}$ is *fully-deterministic*, if $\mathcal{M}$ is deterministic and, for any $x \in X$ and any $q \in Q$ there is at *most* one $\phi \in \text{dom}(F(q))$ that is defined for x. $\diamond$

Definition 4.1.7 An X-machine, $\mathcal{M}$ is *complete w.r.t.* X if for any $x \in X$ and any $q \in Q$, there is at *least* one $\phi \in \text{dom}(F(q))$ that is defined for x. $\diamond$

Definition 4.1.8 An X-machine, $\mathcal{M}$ is *complete w.r.t.* Φ if for all $q \in Q$, $\Phi = \text{dom}(F(q))$. (*i.e.* $F : Q \to (\Phi \to Q)$.) $\diamond$

Lemma 4.1.9 If $\mathcal{M}$ is a fully-deterministic X-machine that is complete w.r.t. X, then, for each state, q, the domains of the $\phi \in \text{dom}(F(q))$ form a partition of X. In other words, suppose $\text{dom}(F(q)) = \{\phi_1, \phi_2, \ldots, \phi_n\}$, then $\text{dom}(\phi_1) \cup \text{dom}(\phi_2) \cup \cdots \cup \text{dom}(\phi_n) = X$, and $\text{dom}(\phi_i) \cap \text{dom}(\phi_j) = \varnothing$ for all $i, j \in \{1..n\}$ where $i \neq j$.

Proof Follows directly from the definitions. $\square$

4.2 Input/Output and X-machines

There are several ways of handling input/output with X-machines.

4.2.1 Input/Output by resolving non-determinism

One technique is to build X-machines such that they are not deterministic, and the "input" is used to resolve the non-determinism.

Example 4.2.1 Consider the following X-machine, $\mathcal{M}_{4.2.1}$:

1. $X = \{a, \ldots, z\}^*$.
2. $Y = \varnothing \qquad Z = X$.
3. $\alpha = \lambda\, y \bullet \langle\rangle$.
4. $\beta = \lambda\, x \bullet x$.
5. $Q = \{\textsc{Read}, \textsc{End}\}$.
6. $\Phi = \{\mathsf{a}, \ldots, \mathsf{z}, \mathsf{a}', \ldots, \mathsf{z}', \mathsf{a}'', \ldots, \mathsf{z}'', \mathsf{Return}\}$ where

$$n = \text{some fixed integer constant}$$

$$\mathsf{a} = \lambda\, x \bullet \begin{cases} x \frown \langle a\rangle & \text{if } \operatorname{len}(x) < n-1 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{b} = \lambda\, x \bullet \begin{cases} x \frown \langle b\rangle & \text{if } \operatorname{len}(x) < n-1 \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

$$\mathsf{z} = \lambda\, x \bullet \begin{cases} x \frown \langle z\rangle & \text{if } \operatorname{len}(x) < n-1 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{Return} = \lambda\, x \bullet x$$

$$\mathsf{a}' = \lambda\, x \bullet \begin{cases} x \frown \langle a\rangle & \text{if } \operatorname{len}(x) = n-1 \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

$$\mathsf{z}' = \lambda\, x \bullet \begin{cases} x \frown \langle a\rangle & \text{if } \operatorname{len}(x) = n-1 \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{a}'' = \lambda\, x \bullet \begin{cases} x & \text{if } \operatorname{len}(x) > n-1 \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

$$\mathsf{z}'' = \lambda\, x \bullet \begin{cases} x & \text{if } \operatorname{len}(x) > n-1 \\ \bot & \text{otherwise} \end{cases}$$

7. For F see figure 4.1.
8. $I = \{\textsc{Read}\}$.

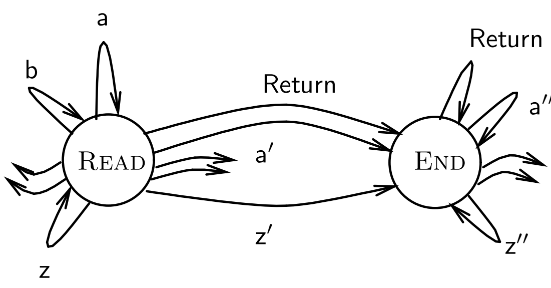


Figure 4.1: A simple X-machine, for entering a text string

9. $T = \{\textsc{End}\}$.

Notice that α and Y are trivial. This is typical of the approach. Also, $\mathcal{M}_{4.2.1}$ is complete w.r.t. X and deterministic, but not complete w.r.t. Φ or fully-deterministic.

The machine models the input of a text string of up to n characters, terminated by a *Return* if it is shorter than n characters. When the machine is in, say, state Read, with fewer than $n-1$ characters entered so far, $\text{dom}(F(\textsc{Read})) = \{\mathsf{a}, \ldots, \mathsf{z}, \mathsf{a}', \ldots, \mathsf{z}', \mathsf{Return}\}$. Of these, any of $\mathsf{a}, \ldots, \mathsf{z}, \mathsf{Return}$ are defined, and so could be the next transition taken. This non-determinism is resolved according to the actual input: if “a” is entered, then transition a is the one taken, and so forth. ◇

One problem with this approach is the large number of very similar transition functions that have to be defined; typically one for every possible input. Even in the case of a trivial example such as $\mathcal{M}_{4.2.1}$ there are three sets of 26 transition functions, that are essentially the same. If input from a mouse were to be modelled, for instance, then one transition function would be needed for *every* screen location the mouse could point at.

4.2.2 Input/Output using streams

A solution to the problem was suggested by Eilenberg in his original discussion of the X-machine model. A typical X would take the form $\Gamma^* \times M \times \Sigma^*$. Here, Σ^* is the *input stream*, formed of sequences from the alphabet Σ, and Γ^* the *output stream*, formed of sequences from the alphabet Γ. M is the *memory* of the machine.

Suppose $x = (G, m, S)$, where $S \in \Sigma^*$, $m \in M$ and $G \in \Gamma^*$. Interpret S as the sequence of unprocessed input so far, and G as the sequence of output, so far. The next input is *head* S, and the next transition function will remove it, update m, and add a new output to the head of G.

The output sequence can be considered as “instructions” to the output device(s), which are interpreted by β. Or, alternatively, the output can be considered as a sequence of “snap-shots” of the output status.

Example 4.2.2 Consider the following X-machine, $\mathcal{M}_{4.2.2}$:

1. $X = \Gamma^* \times M \times \Sigma^*$, where

$$\begin{array}{ll} \Sigma = \{a, \ldots, z, \mathit{Return}\} & \Gamma = \{a, \ldots, z\} \\ M = \{a, \ldots, z\}^* & \end{array}$$

2. $Y = \Sigma^*$.

3. $Z = \Gamma^*$.

4. $\alpha = \lambda\, S \bullet (\langle\rangle, \langle\rangle, S)$.

5. $\beta = \lambda(G, m, S) \bullet G$.

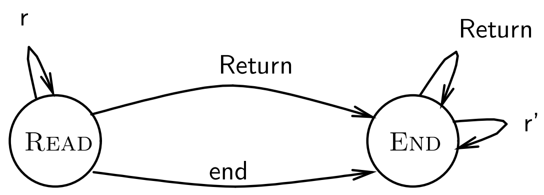


Figure 4.2: An X-machine using general functions.

6. $Q = \{\textsc{Read}, \textsc{End}\}$.

7. $\Phi = \{\mathsf{r}, \mathsf{Return}, \mathsf{end}, \mathsf{r}'\}$, where

$$n = \text{some fixed integer constant}$$

$$\mathsf{r} = \lambda(G, m, S) \bullet \begin{cases} (G \frown \mathit{head}\, S, & \\ \quad m \frown \mathit{head}\, S, \mathit{tail}\, S) & \text{if } \mathit{head}\, S \in \{a, \ldots z\} \wedge \\ & \text{len}(m) < n - 1 \\ \perp & \text{otherwise} \end{cases}$$

$$\mathsf{r}' = \lambda(G, m, S) \bullet \begin{cases} (G, m, \mathit{tail}\, S) & \text{if } \mathit{head}\, S \in \{a, \ldots, z\} \wedge \\ & \text{len}(m) > n - 1 \\ \perp & \text{otherwise} \end{cases}$$

$$\mathsf{end} = \lambda(G, m, S) \bullet \begin{cases} (G \frown \mathit{head}\, S, & \\ \quad m \frown \mathit{head}\, S, \mathit{tail}\, S) & \text{if } \mathit{head}\, S \in \{a, \ldots z\} \wedge \\ & \wedge\ \text{len}(m) = n - 1 \\ \perp & \text{otherwise} \end{cases}$$

$$\mathsf{Return} = \lambda(G, m, S) \bullet \begin{cases} (G, m, \mathit{tail}\, S) & \text{if } \mathit{head}\, S = \mathit{Return} \\ \perp & \text{otherwise} \end{cases}$$

8. For F, see figure 4.2.

9. $I = \{\textsc{Read}\}$.

10. $T = \{\textsc{End}\}$.

Notice that α and β are still fairly simple. $\mathcal{M}_{4.2.2}$ is complete w.r.t. X and fully-deterministic, but not complete w.r.t. Φ.

$\mathcal{M}_{4.2.2}$ models the same string input behaviour as $\mathcal{M}_{4.2.1}$, but uses far fewer functions. $\diamond$

Examples 4.2.1 and 4.2.2 represent two extremes in the handling of input to X-machines, but there is no reason why the approaches cannot be combined when it is appropriate.

Definition 4.2.3 An X-machine with $X = \Gamma^* \times M \times \Sigma^*$, is a *stream-*X*-machine* if

1. there exists a bijection, $\alpha^* : Y \to \Sigma^*$ such that $\alpha = \lambda\, y \bullet (\langle\rangle, m_0, \alpha^*(y))$ where m_0 is the *initial memory* value of the stream-X-machine.

2. there exists a bijection, $\beta^* : \Gamma^* \to Z$ such that $\beta = \lambda(G, m, S) \bullet \beta^*(G)$.

The M component of X is known as the *memory* of the machine.

Each transition function must remove the head of the input stream and add a single element to the head of the output stream, and, furthermore, no transition is allowed to use information from the tail of the input or any of the output. *i.e.* every $\phi \in \Phi$ is of the following form:

$$\phi = \lambda(g, m, h :: s) : \Gamma^* \times M \times \Sigma^* \bullet \begin{cases} (\gamma_1(m,h) :: g, \phi_1(m,h), s) & \text{if } c_1(m,h) \\ (\gamma_2(m,h) :: g, \phi_2(m,h), s) & \text{if } c_2(m,h) \\ \quad\vdots & \\ \quad\perp & \text{otherwise} \end{cases}$$

where the ϕ_is are functions of m and h (*i.e.* $\phi_i : M \times \Sigma \to M$), the γ_i are functions of m and h (*i.e.* $\gamma_i : M \times \Sigma \to \Gamma$) and the c_is are mutually exclusive conditions on m and h (*i.e.* $c_i : M \times \Sigma \to \mathbb{B}$). ◇

The machine in example 4.2.2 ($\mathcal{M}_{4.2.2}$) is a stream-X-machine.

Remark 4.2.4 Part of the motivation behind the definition of stream-X-machines is so that they have the following property: There is a concatenation operator, $\smile$, on Y so that the following diagram commutes:

$$\begin{tikzcd} Y \times Y \arrow[r, "\smile"] \arrow[d, "\alpha^*"'] & Y \arrow[d, "\alpha^*"] \\ \Sigma^* \times \Sigma^* \arrow[r, "\frown"] & \Sigma^* \end{tikzcd}$$

Since input is converted to Σ^* by a bijection (α^*), input may often be considered to be from the set Σ^* as from the set Y. Thus the phrase "input sequence $S \in \Sigma^*$" is meaningful. ◇

Definition 4.2.5 A stream-X-machine, $\mathcal{M}$, with $X = \Gamma^* \times M \times \Sigma^*$, as above, is *complete w.r.t.* Y if, for any $y \in Y$, there is *at least* one $q \in I$ and path, p, with $\text{len}(p_\Phi) = \text{len}(\alpha^*(y))$, and starting at q, such that $\beta(|p|(\alpha(y)))$ is defined. ◇

Definition 4.2.6 Given a stream-X-machine, $\mathcal{M}$, with $X = \Gamma^* \times M \times \Sigma^*$, there are projection functions as follows:

$$\begin{aligned} \text{Output} &= \lambda(G, m, S) : X \bullet G \\ \text{Mem} &= \lambda(G, m, S) : X \bullet m \\ \text{Input} &= \lambda(G, m, S) : X \bullet S \\ \text{Mem\_Input} &= \lambda(G, m, S) : X \bullet (m, \textit{head}\ S) \end{aligned}$$

◇

Definition 4.2.7 Given a stream-X-machine, $\mathcal{M}$, $m \in M$ and $q \in Q$, m is *attainable* in q if there is an input sequence, $y \in Y$, and path p such that $q_a \xrightarrow{p} q$ (where $q_a \in I$) and $m = \text{Mem}(|p|(\alpha(y)))$. ◇

Definition 4.2.8 Given a stream-X-machine, $\mathcal{M}$, with a state $q \in Q$,

$$\begin{aligned} \text{Attainable}(q) &= \{(m, s) : M \times \Sigma \mid m \text{ is attainable in } q, s \in \Sigma\}, \\ \text{MAttainable}(q) &= \{m : M \mid m \text{ is attainable in } q\}, \\ \text{XAttainable}(q) &= \{x : X \mid \text{Mem\_Input}(x) \in \text{Attainable}(q)\} \end{aligned}$$

are the sets of values from $M \times \Sigma$, M and X respectively that can occur in state q. ◇

Aside When discussing the internal behaviour of a stream-X-machine, it is often convenient to talk about partial paths, starting at an arbitrary state $q \in Q$ (with q not necessarily in I), with an arbitrary (partial) input sequence $S \in \Sigma^*$, and an arbitrary initial value for the memory, $m \in \text{MAttainable}(q)$. ◇

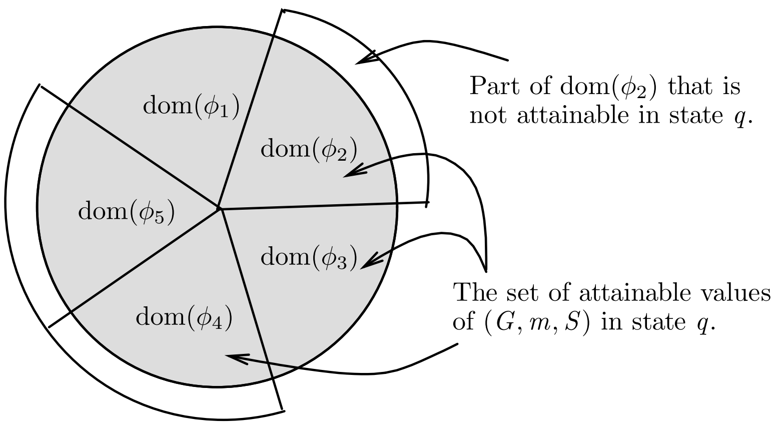

dom(ϕ_1)
dom(ϕ_2)
dom(ϕ_3)
dom(ϕ_4)
dom(ϕ_5)
Part of dom(ϕ_2) that is not attainable in state q.
The set of attainable values of (G, m, S) in state q.

Figure 4.3: Venn diagram showing the relationship between attainable and defined elements of $\Gamma^* M \times \Sigma^*$ for a given state q.

Lemma 4.2.9 If $\mathcal{M}$ is a fully-deterministic stream-X-machine that is complete w.r.t. Y, then, for each state $q \in Q$, the domains of the $\phi \in \text{dom}(F(q))$ form a partition of the set XAttainable(q). In other words, if

$$\text{dom}(F(q)) = \{\phi_1, \ldots, \phi_n\},$$

then

$$\text{dom}(\phi_1) \cup \text{dom}(\phi_2) \cup \cdots \cup \text{dom}(\phi_n) \supseteq \text{XAttainable}(q),$$

and

$$\text{dom}(\phi_i) \cap \text{dom}(\phi_j) = \varnothing$$

for any $i, j \in \{1..n\}$ where $i \neq j$.

Proof Follows directly from the definitions. $\square$

NB A given $\phi \in \text{dom}(F(q))$ may be defined for some values of (G, m, S) which are not in the set XAttainable(q), as illustrated by figure 4.3. $\diamond$

4.3 Some X-machine theory.

4.3.1 Preliminary definitions.

Throughout section 4.3

$$\mathcal{M} = (X, Y, Z, \alpha, \beta, Q_{\mathcal{M}}, \Phi, F_{\mathcal{M}}, I_{\mathcal{M}}, T_{\mathcal{M}})$$
$$\mathcal{N} = (X, Y, Z, \alpha, \beta, Q_{\mathcal{N}}, \Phi, F_{\mathcal{N}}, I_{\mathcal{N}}, T_{\mathcal{N}})$$

$\mathcal{M}$ and $\mathcal{N}$ are both fully-deterministic stream-X-machines that are complete w.r.t. Y, and $X = \Gamma^* \times M \times \Sigma^*$.

Notation 4.3.1 Given $\mathcal{M}$, with state $q \in Q_{\mathcal{M}}$, $m \in$ MAttainable(q), and (partial) input sequence $S \in \Sigma^*$,

- $q \mid_\Phi (m, S) \in \Phi^*$ denotes the Φ-path through $\mathcal{M}$ resulting from a start value of $X = (\langle\rangle, m, S)$ in state q.
- $q \mid_\Gamma (m, S) \in \Gamma^*$ denotes the output sequence of $\mathcal{M}$, resulting from a start value of $X = (\langle\rangle, m, S)$, in state q.
- $q \mid_Q (m, S) \in Q^*$ denotes the sequence of states passed through in $\mathcal{M}$, (*i.e.* the Q-path) from a start value of $X = (\langle\rangle, m, S)$, in state q.
- $q \mid_M (m, S) \in M^*$ denotes the sequence of values of M that are taken during the course of $\mathcal{M}$'s operation, with start value $X = (\langle\rangle, m, S)$, in state q.
- $q \mid_X (m, S) \in X^*$ denotes the sequence of values of X that are taken by $\mathcal{M}$ in the course of its operation, with start value $X = (\langle\rangle, m, S)$, in state q.

The sequences are all built up so that the "most recent" element is the head, apart from the sequence of ϕ's, which has its most recent element last. ◇

Notation 4.3.2 Given $\mathcal{M}$, with state $q \in Q_{\mathcal{M}}$, $m \in \text{MAttainable}(q)$, and (partial) input sequence $S \in \Sigma^*$,

- $q \parallel_\Phi (m, S) \in \Phi$ denotes the final transition followed in the operation of $\mathcal{M}$ with start value $X = (\langle\rangle, m, S)$ in state q.

  $$q \parallel_\Phi (m, S) = \textit{last}\ q \mid_\Phi (m, S).$$

- $q \parallel_\Gamma (m, S) \in \Gamma$ denotes the final output of the operation of $\mathcal{M}$ with start value $X = (\langle\rangle, m, S)$ in state q.

  $$q \parallel_\Gamma (m, S) = \textit{head}\ q \mid_\Gamma (m, S).$$

- $q \parallel_Q (m, S) \in Q$ denotes the final state reached in $\mathcal{M}$ with start value $X = (\langle\rangle, m, S)$ in state q.

  $$q \parallel_Q (m, S) = \textit{head}\ q \mid_Q (m, S).$$

- $q \parallel_M (m, S) \in M$ denotes the final value of M taken in the operation of $\mathcal{M}$ with start value $X = (\langle\rangle, m, S)$ in state q.

  $$q \parallel_M (m, S) = \textit{head}\ q \mid_M (m, S).$$

- $q \parallel_X (m, S) \in X$ denotes the final value of X taken in the operation of $\mathcal{M}$ with start value $X = (\langle\rangle, m, S)$ in state q.

  $$q \parallel_X (m, S) = \textit{head}\ q \mid_X (m, S).$$

  Also,

  $$q \parallel_X (m, S) = (q \mid_\Gamma (m, S),\ q \parallel_M (m, S),\ \langle\rangle).$$

◇

Observation 4.3.3

$$\text{len}(S) = \text{len}(q \mid_\Phi (m, S)) = \text{len}(q \mid_\Gamma (m, S)) = \text{len}(q \mid_Q (m, S)) - 1.$$

Observation 4.3.4 The stream-X-machine definition was, in part, motivated by the following properties (which are described in terms of fully-deterministic stream-X-machine, $\mathcal{M}$ that is complete w.r.t. Y).

If

$$q_0 \parallel_M (m_0, S_1) = m_1$$
$$q_0 \parallel_Q (m_0, S_1) = q_1$$

then

$$[q_0 \mid_\Phi (m_0, S_1)] \frown [q_1 \mid_\Phi (m_1, S_2)] = q_0 \mid_\Phi (m_0, S_1 \frown S_2)$$
$$[q_0 \mid_\Gamma (m_0, S_1)] \frown [q_1 \mid_\Gamma (m_1, S_2)] = q_0 \mid_\Gamma (m_0, S_1 \frown S_2)$$

$\diamond$

Notation 4.3.5 In cases where there is ambiguity regarding which machine a state belongs to, the following convention is used. Suppose X-machines $\mathcal{M}$ and $\mathcal{N}$ both have a state called q. Then, $q^{\mathcal{M}}$ belongs to $\mathcal{M}$, and $q^{\mathcal{N}}$ belongs to $\mathcal{N}$. $\diamond$

Definition 4.3.6 A state $q \in Q$ is *reachable* in $\mathcal{M}$ iff

$$\exists\, S \in \Sigma^* \bullet q_0 \parallel_Q (m_0, S) = q.$$

$\diamond$

4.3.2 Equivalence and minimality of X-machines.

Definition 4.3.7 Given states q_1 and q_2 from machines of the same type, Φ, then q_1 and q_2 are Φ-*equivalent* iff

$$\text{MAttainable}(q_1) = \text{MAttainable}(q_2) \wedge$$
$$\forall\, m \in \text{MAttainable}(q_1);\ \forall\, S \in \Sigma^* \bullet q_1 \mid_\Phi (m, S) = q_2 \mid_\Phi (m, S),$$

and write $q_1 \equiv_\Phi q_2$. $\diamond$

Definition 4.3.8 Given $\mathcal{M}$, with $Q_{\mathcal{M}} = \{q_0, \ldots, q_n\}$, $\mathcal{M}$ is Φ-*minimal* iff

$$\forall\, q_i, q_j \in Q_{\mathcal{M}} \bullet (q_i \neq q_j) \Rightarrow \neg\,(q_i \equiv_\Phi q_j).$$

$\diamond$

Definition 4.3.9 Given $\mathcal{M}$ and $\mathcal{N}$, they are Φ-*equivalent* iff their initial states (p_0 and q_0) are Φ-equivalent, *i.e.* $p_0 \equiv_\Phi q_0$, and write $\mathcal{M} \equiv_\Phi \mathcal{N}$. $\diamond$

Definition 4.3.10 An X-machine $\mathcal{M}$ is Q-*minimal* if it has fewer states than any X-machine $\mathcal{N}$ that is Φ-equivalent to it. $\diamond$

Lemma 4.3.11 If an X-machine $\mathcal{M}$ is Q-minimal then every state is reachable.

Proof This follows directly, as the unreachable states can be removed without affecting the possible paths in $\mathcal{M}$. $\square$

Theorem 4.3.12 $\mathcal{M}$ is Q-minimal (as in 4.3.10) $\Leftrightarrow$ it is Φ-minimal (as in 4.3.8) and every state is reachable.

Proof

$\Rightarrow$ Suppose $\mathcal{M}$ is Q-minimal, and show that it must also be Φ-minimal, with every state reachable.

Firstly, since $\mathcal{M}$ is Q-minimal, every state must be reachable.

Use a proof by contradiction to show that $\mathcal{M}$ is Φ-minimal: suppose $\mathcal{M}$ is not Φ-minimal. Then

$$\exists\, q_1, q_2 \in Q_{\mathcal{M}} \bullet (q_1 \neq q_2) \wedge (q_1 \equiv_\Phi q_2).$$

Without loss of generality, $q_2 \neq q_0^{\mathcal{M}}$.

Construct $\mathcal{N}$, of type Φ, which is exactly the same as $\mathcal{M}$, except that it has no state q_2. Instead, for every $(q, \phi) \in (Q_{\mathcal{N}} \times \Phi)$ such that $F_{\mathcal{M}}(q, \phi) = q_2$, put $F_{\mathcal{N}}(q, \phi) = q_1$.

Consider the initial states of the two X-machines, $q_0^{\mathcal{M}}$ and $q_0^{\mathcal{N}}$ respectively, and any $y \in Y$, with

$$\alpha(y) = (\langle\rangle, m_0, S),$$

which is the same for $\mathcal{M}$ and $\mathcal{N}$.

Now, either $q_2 \in q_0^{\mathcal{M}} \mid_Q (m_0, S)$ or $q_2 \notin q_0^{\mathcal{M}} \mid_Q (m_0, S)$.

1. If $q_2 \notin q_0 \mid_Q (m_0, S)$, then clearly

   $$q_0^{\mathcal{M}} \mid_\Phi (m_0, S) = q_0^{\mathcal{N}} \mid_\Phi (m_0, S),$$

   since the two machines are exactly the same apart from q_2.

2. If $q_2 \in q_0 \mid_Q (m_0, S)$, then S can be split into two sections, $S = S' \frown S''$ such that

   $$q_0^{\mathcal{M}} \parallel_Q (m_0, S') = q_2,$$

   and

   $$q_0^{\mathcal{M}} \parallel_M (m_0, S') = m',$$
   $$q_2 \notin q_2 \mid_Q (m', S'').$$

   Also, $q_0^{\mathcal{N}} \parallel_Q (m_0, S') = q_1$, from the definition of $\mathcal{N}$.

   Repeat the argument on m' and S' instead of m_0 and S as many times as necessary.

So, $q_0^{\mathcal{M}} \mid_\Phi (m_0, S) = q_0^{\mathcal{N}} \mid_\Phi (m_0, S)$, and $q_0^{\mathcal{M}} \equiv_\Phi q_0^{\mathcal{N}}$, but $\mathcal{N}$ has fewer states than $\mathcal{M}$, which contradicts the Q-minimality of $\mathcal{M}$.

Therefore, $\mathcal{M}$ is Φ-minimal.

$\Leftarrow$ Suppose $\mathcal{M}$ is Φ-minimal with every state reachable and show that it must be Q-minimal.

Then

$$\forall\, q_1, q_2 \in Q_{\mathcal{M}} \bullet (q_1 \neq q_2) \Rightarrow \neg\, (q_1 \equiv_\Phi q_2).$$

Use a proof by contradiction: suppose that $\mathcal{M}$ is not Q-minimal. Then $\exists \mathcal{N}$ of type Φ with $\mathcal{N} \equiv_\Phi \mathcal{M}$, but containing fewer states. In fact, w.l.o.g. it can be assumed that $\mathcal{N}$ is Q-minimal.

Consider any $q^{\mathcal{M}} \in Q_{\mathcal{M}}$. Every state is reachable, so there is at least one input sequence leading to $q^{\mathcal{M}}$. Consider any $m \in \text{MAttainable}(q^{\mathcal{M}})$; then $\exists\, S' \in \Sigma^*$ such that $q_0^{\mathcal{M}} \parallel_Q (m_0, S') = q^{\mathcal{M}}$, and $m = q_0 \parallel_M (m_0, S')$.

Let $q^{\mathcal{N}} = q_0^{\mathcal{N}} \parallel_Q (m, S')$. Since $\mathcal{N} \equiv_\Phi \mathcal{M}$,

$$q_0^{\mathcal{M}} \mid_\Phi (m_0, S') = q_0^{\mathcal{N}} \mid_\Phi (m_0, S'),$$

and therefore,

$$q_0^{\mathcal{M}} \parallel_M (m_0, S') = q_0^{\mathcal{N}} \parallel_M (m_0, S') = m.$$

Also, since $\mathcal{M} \equiv_\Phi \mathcal{N}$,

$$\forall\, S'' \in \Sigma^* \bullet q_0^{\mathcal{N}} \mid_\Phi (m_0, S' \frown S'') = q_0^{\mathcal{M}} \mid_\Phi (m_0, S' \frown S'').$$

Therefore,

$$q^{\mathcal{N}} \mid_\Phi (m, S'') = q^{\mathcal{M}} \mid_\Phi (m, S'').$$

This is true for any $m \in \text{MAttainable}(q^{\mathcal{M}})$ and $S'' \in \Sigma^*$, so $q^{\mathcal{M}} \equiv_\Phi q^{\mathcal{N}}$.

Since there are fewer states in $\mathcal{N}$ than in $\mathcal{M}$, this means that there must be $q^{\mathcal{N}} \in Q_{\mathcal{N}}$ such that

$$\exists\, q_1^{\mathcal{M}}, q_2^{\mathcal{M}} \in Q_{\mathcal{M}} \bullet q_1^{\mathcal{M}} \equiv_\Phi q^{\mathcal{N}} \wedge q_2^{\mathcal{M}} \equiv_\Phi q^{\mathcal{N}} \wedge q_1^{\mathcal{M}} \neq q_2^{\mathcal{N}}.$$

By the transitivity of $\equiv_\Phi$, $q_1^{\mathcal{M}} \equiv_\Phi q_2^{\mathcal{M}}$, which contradicts the Φ-minimality of $\mathcal{M}$.

Therefore, $\mathcal{M}$ is Q-minimal.

□

Definition 4.3.13 Given states q_1 and q_2, then q_1 and q_2 are Γ-*equivalent* iff

$$\text{MAttainable}(q_1) = \text{MAttainable}(q_2) \wedge$$
$$\forall\, m \in \text{MAttainable}(q_1);\ \forall\, S \in \Sigma^* \bullet q_1 \mid_\Gamma (m, S) = q_2 \mid_\Gamma (m, S),$$

and write $q_1 \equiv_\Gamma q_2$. ◇

Definition 4.3.14 Given $\mathcal{M}$, with states $Q_{\mathcal{M}} = \{q_0, \ldots, q_n\}$, $\mathcal{M}$ is Γ-*minimal* iff

$$\forall\, q_i, q_j \in Q_{\mathcal{M}} \bullet (q_1 \neq q_2) \Rightarrow \neg\, (q_i \equiv_\Gamma q_j).$$

◇

Definition 4.3.15 Given $\mathcal{M}$ and $\mathcal{N}$, they are Γ-*equivalent* iff their initial states (p_0 and q_0) are Γ-equivalent, *i.e.* $p_0 \equiv_\Gamma q_0$, and write $\mathcal{M} \equiv_\Gamma \mathcal{N}$. ◇

Definition 4.3.16 An $\mathcal{M}$ is Q-Γ-*minimal* if it has fewer states than any X-machine $\mathcal{N}$ that is Γ-equivalent to it. ◇

Lemma 4.3.17 If $\mathcal{M}$ is Q-Γ-minimal, then every state is reachable.

Proof If a state is not reachable, then it can be removed to produce a Γ-equivalent machine that has fewer states. □

Theorem 4.3.18 *X-machine $\mathcal{M}$ is Q-Γ-minimal (as in 4.3.16) $\Leftrightarrow$ it is Γ-minimal (as in 4.3.14) and every state is reachable.*

Proof This is very similar to the proof of Theorem 4.3.12.

$\Rightarrow$ Suppose $\mathcal{M}$ is Q-Γ-minimal, and show that it must also be Γ-minimal with every state reachable.

Firstly, $\mathcal{M}$ is Q-Γ-minimal, so every state is reachable.

Use a proof by contradiction to show that it must also be Γ-minimal: suppose $\mathcal{M}$ is not Γ-minimal. Then

$$\exists\, q_1, q_2 \in Q_{\mathcal{M}} \bullet (q_1 \neq q_2) \wedge (q_1 \equiv_\Gamma q_2).$$

Without loss of generality, $q_2 \neq q_0^{\mathcal{M}}$.

Construct $\mathcal{N}$, of type Φ, which is exactly the same as $\mathcal{M}$, except that it has no state q_2. Instead, for every $(q, \phi) \in (Q_{\mathcal{N}} \times \Phi)$ such that $F_{\mathcal{M}}(q, \phi) = q_2$, put $F_{\mathcal{N}}(q, \phi) = q_1$.

Consider the initial states of the two X-machines, $q_0^{\mathcal{M}}$ and $q_0^{\mathcal{N}}$ respectively, and any $y \in Y$, with

$$\alpha(y) = (\langle\rangle, m_0, S),$$

which is the same for $\mathcal{M}$ and $\mathcal{N}$.

Now, either $q_2 \in q_0^{\mathcal{M}} \mid_Q (m_0, S)$ or $q_2 \notin q_0^{\mathcal{M}} \mid_Q (m_0, S)$.

1. If $q_2 \notin q_0 \mid_Q (m_0, S)$, then clearly

$$q_0^{\mathcal{M}} \mid_\Gamma (m_0, S) = q_0^{\mathcal{N}} \mid_\Gamma (m_0, S),$$

since the two machines are exactly the same apart from q_2.

2. If $q_2 \in q_0 \mid_Q (m_0, S)$, then S can be split into two sections, $S = S' \frown S''$ such that

$$q_0^{\mathcal{M}} \parallel_Q (m_0, S') = q_2,$$

and

$$q_0^{\mathcal{M}} \parallel_M (m_0, S') = m',$$
$$q_2 \notin q_2 \mid_Q (m', S'').$$

Also, $q_0^{\mathcal{N}} \parallel_Q (m_0, S') = q_1$, from the definition of $\mathcal{N}$.
Repeat the argument on m' and S' instead of m_0 and S as many times as necessary.

So, $q_0^{\mathcal{M}} \mid_\Gamma (m_0, S) = q_0^{\mathcal{N}} \mid_\Gamma (m_0, S)$, and $q_0^{\mathcal{M}} \equiv_\Gamma q_0^{\mathcal{N}}$, but $\mathcal{N}$ has fewer states than $\mathcal{M}$, which contradicts the Q-Γ-minimality of $\mathcal{M}$.

Therefore, $\mathcal{M}$ is Γ-minimal.

$\Leftarrow$ Suppose $\mathcal{M}$ is Γ-minimal with every state reachable and show that it must be Q-Γ-minimal.
Then

$$\forall\, q_1, q_2 \in Q_{\mathcal{M}} \bullet (q_1 \neq q_2) \Rightarrow \neg\,(q_1 \equiv_\Gamma q_2).$$

Use a proof by contradiction: suppose that $\mathcal{M}$ is not Q-Γ-minimal. Then $\exists \mathcal{N}$ of type Φ with $\mathcal{N} \equiv_\Gamma \mathcal{M}$, but containing fewer states. In fact, w.l.o.g. it can be assumed that $\mathcal{N}$ is Q-Γ-minimal.

Consider any $q^{\mathcal{M}} \in Q_{\mathcal{M}}$. Every state is reachable, so there is at least one input sequence leading to $q^{\mathcal{M}}$. Consider any $m \in \text{MAttainable}(q^{\mathcal{M}})$; then $\exists\, S' \in \Sigma^*$ such that $q_0^{\mathcal{M}} \parallel_Q (m_0, S') = q^{\mathcal{M}}$, and $m = q_0 \parallel_M (m_0, S')$.

Let $q^{\mathcal{N}} = q_0^{\mathcal{N}} \parallel_Q (m, S')$. Since $\mathcal{N} \equiv_\Gamma \mathcal{M}$,

$$q_0^{\mathcal{M}} \mid_\Gamma (m_0, S') = q_0^{\mathcal{N}} \mid_\Gamma (m_0, S'),$$

and therefore,

$$q_0^{\mathcal{M}} \parallel_M (m_0, S') = q_0^{\mathcal{N}} \parallel_M (m_0, S') = m.$$

Also, since $\mathcal{M} \equiv_\Gamma \mathcal{N}$,

$$\forall\, S'' \in \Sigma^* \bullet q_0^{\mathcal{N}} \mid_\Gamma (m_0, S' \frown S'') = q_0^{\mathcal{M}} \mid_\Gamma (m_0, S' \frown S'').$$

Therefore,

$$q^{\mathcal{N}} \mid_\Gamma (m, S'') = q^{\mathcal{M}} \mid_\Gamma (m, S'').$$

This is true for any $m \in \text{MAttainable}(q^{\mathcal{M}})$ and $S'' \in \Sigma^*$, so $q^{\mathcal{M}} \equiv_\Gamma q^{\mathcal{N}}$.

Since there are fewer states in $\mathcal{N}$ than in $\mathcal{M}$, this means that there must be $q^{\mathcal{N}} \in Q_{\mathcal{N}}$ such that

$$\exists\, q_1^{\mathcal{M}}, q_2^{\mathcal{M}} \in Q_{\mathcal{M}} \bullet q_1^{\mathcal{M}} \equiv_\Gamma q^{\mathcal{N}} \wedge q_2^{\mathcal{M}} \equiv_\Gamma q^{\mathcal{N}} \wedge q_1^{\mathcal{M}} \neq q_2^{\mathcal{N}}.$$

By the transitivity of $\equiv_\Gamma$, $q_1^{\mathcal{M}} \equiv_\Gamma q_2^{\mathcal{M}}$, which contradicts the Γ-minimality of $\mathcal{M}$.

Therefore, $\mathcal{M}$ is Q-Γ-minimal.

□

4.4 Refinement of X-machines.

If X-machines are to be useful as a tool for specification, there needs to be a way to develop existing machines into more complex and detailed versions, without having to start anew with each modification.

4.4.1 Simple refinements.

The most straight-forward type of refinement of an X-machine is to replace one of its states, q say, with an "X-machine-let", consisting of several new states, and new arcs joining them to q.

Definition 4.4.1 An X-*machine-let* is an X-machine with a single initial state, and a single terminal state. The terminal state is the same as the initial state. ◇

The operation of the X-machine-let is defined in the same way as for an ordinary X-machine. Similarly, definitions 4.1.2 through to 4.2.5 have direct analogues for X-machine-lets.

Definition 4.4.2 Suppose $\mathcal{M}$ is a fully-deterministic X-machine, with

$$\mathcal{M} = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T),$$

and $\mathcal{N}$ is a fully-deterministic X-machine-let, with

$$\mathcal{N} = (X, Y, Z, \alpha, \beta, Q_{\mathcal{N}}, \Phi_{\mathcal{N}}, F_{\mathcal{N}}, \{q\}, \{q\}),$$

and $q \in Q \cap Q_{\mathcal{N}}$. Further, for any $x \in X$ such that $\exists\, \phi \in \operatorname{dom}(F(q))$ for which $\phi(x)$ is defined, there must be a $\phi' \in \operatorname{dom}(F_{\mathcal{N}}(q))$ such that $\phi'(x)$ is defined. Then X-machine $\mathcal{M}'$ (of type Φ') is the *simple-refinement* of $\mathcal{M}$ using $\mathcal{N}$ if:

$$\mathcal{M}' = (X', Y', Z', \alpha', \beta', Q', \Phi', F', I', T'),$$

where

$$\begin{aligned}
X' &= X\\
Y' &= Y\\
Z' &= Z\\
\alpha' &= \alpha\\
\beta' &= \beta\\
Q' &= Q \cup Q_{\mathcal{N}}\\
&\quad Q \cap Q_{\mathcal{N}} = \{q\}\\
\Phi' &= \Phi \cup \Phi_{\mathcal{N}}\\
&\quad \Phi \cap \Phi_{\mathcal{N}} \supseteq \operatorname{dom}(F(q) \rhd \{q\})\\
F' &= \lambda\, p : Q' \bullet \begin{cases} F(p) & \text{if } p \in (Q \setminus \{q\})\\ F_{\mathcal{N}}(p) & \text{if } p \in (Q_{\mathcal{N}} \setminus \{q\})\\ f_q & \text{if } p = q \end{cases}\\
I' &= I\\
T' &= T\\
I_{\mathcal{N}} &= q
\end{aligned}$$

and f_q is defined as follows:

$$f_q = F_{\mathcal{N}}(q) \oplus (F(q) \rhd \{q\})$$

◇

Informally, a refinement of $\mathcal{M}$ using $\mathcal{N}$ to produce $\mathcal{M}'$, can be described as follows:

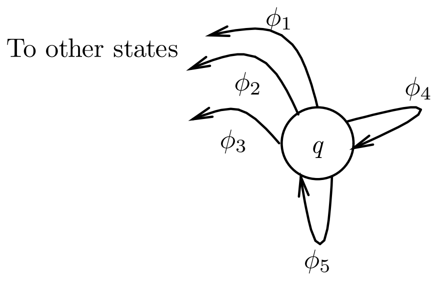


Figure 4.4: Diagram showing $F(q)$ for $\mathcal{M}$.

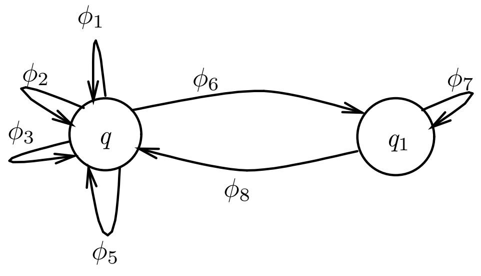


Figure 4.5: The X-machine-let, $\mathcal{N}$.

1. $\mathcal{M}$ and $\mathcal{N}$ have state q in common.
2. All the transitions of q in $\mathcal{M}$ that lead to other states must be present as "loop-back" transitions of q (*i.e.* of the form $q \xrightarrow{\phi} q$) in $\mathcal{N}$. Furthermore, these transitions are the same in $\mathcal{M}'$ as in $\mathcal{M}$.
3. Transitions that are present as "loop-backs" in both $\mathcal{M}$ and $\mathcal{N}$ are present in $\mathcal{M}'$.
4. Transitions that are "loop-backs" in $\mathcal{M}$ but **not** in $\mathcal{N}$ are not present in $\mathcal{M}'$. They are replaced by the transitions of q in $\mathcal{N}$ that lead to other states.

Example 4.4.3 Consider X-machine $\mathcal{M}$, containing state q which is as shown in figure 4.4, and X-machine-let $\mathcal{N}$, also containing state q, as shown in figure 4.5. Then $\mathcal{M}'$, the Q-simple refinement of $\mathcal{M}$ using $\mathcal{N}$ is as shown in figure 4.6.

$$\begin{aligned} F(q) &= \{\phi_1 \mapsto q_a, \phi_2 \mapsto q_b, \phi_3 \mapsto q_c, \phi_4 \mapsto q, \phi_5 \mapsto q\} \\ F_{\mathcal{N}}(q) &= \{\phi_1 \mapsto q, \phi_2 \mapsto q, \phi_3 \mapsto q, \phi_5 \mapsto q, \phi_6 \mapsto q_1\} \end{aligned}$$

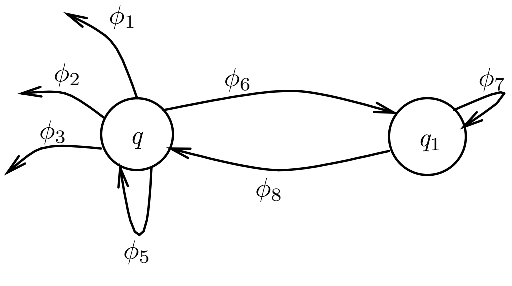


Figure 4.6: Diagram showing the "new" part of $\mathcal{M}'$, the Q-simple refinement of $\mathcal{M}$ using $\mathcal{N}$.

so

$$F(q) \rhd \{q\} = \{\phi_1 \mapsto q_a, \phi_2 \mapsto q_b, \phi_3 \mapsto q_c\}$$

and therefore

$$\begin{aligned} f_q &= F_{\mathcal{N}}(q) \oplus (F(q) \rhd \{q\}) \\ &= \{\phi_1 \mapsto q_a, \phi_2 \mapsto q_b, \phi_3 \mapsto q_c, \phi_5 \mapsto q, \phi_6 \mapsto q_1\}. \end{aligned}$$

◇

Lemma 4.4.4 For a fully-deterministic stream-X-machine that is complete w.r.t. Y, with $S \in \Sigma^*$ and a path p through the machine starting in state q, and where $G \in \Gamma^*$, $m \in \text{MAttainable}(q)$, let

$$(G', m', S') = |p|\,(G, m, S).$$

Then $\text{len}(p_\Phi) = \text{len}(S) - \text{len}(S')$.

Proof Follows directly from the definition of a stream-X-machine (definition 4.2.3). □

Theorem 4.4.5 Suppose $\mathcal{M}$ is a stream-X-machine, and $\mathcal{N}$ a stream-X-machine-let with q in common with $\mathcal{M}$, $\mathcal{M}$ and $\mathcal{N}$ are both fully-deterministic and complete w.r.t. Y, and $\mathcal{M}$ is Q-Γ-minimal.

$$\begin{aligned} \mathcal{M} &= (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T) \\ \mathcal{N} &= (X, Y, Z, \alpha, \beta, Q_{\mathcal{N}}, \Phi_{\mathcal{N}}, F_{\mathcal{N}}, \{q\}, \{q\}) \end{aligned}$$

Then $\mathcal{M}'$, the Q-simple-refinement of $\mathcal{M}$ using $\mathcal{N}$, is a stream-X-machine that is fully-deterministic and complete w.r.t. Y.

Proof It is clear that $\mathcal{M}'$ is a stream-X-machine, so it remains to show that $\mathcal{M}'$ is fully-deterministic and complete w.r.t. Y.

fully-deterministic: Firstly, $\mathcal{M}'$ is deterministic, since:

- α' and β' are both functions;
- Φ and $\Phi_{\mathcal{N}}$ both contain only functions on X, so Φ' contains only functions.
- $F'(p) : \Phi' \nrightarrow Q'$ for all $p \in Q'$.

Secondly, it is required that for any $x \in X$, and $p \in Q$ there be at most one $\phi \in F(p)$ that is defined.

If $p \in (Q \setminus \{q\})$, then $F'(p) = F(p)$, and there is at most one $\phi \in F'(p)$ that is defined for x.

Similarly, if $p \in (Q_{\mathcal{N}} \setminus \{q\})$, then $F'(p) = F_{\mathcal{N}}(p)$, and there is at most one $\phi \in F'(p)$ that is defined for x.

If $p = q$, then consider $F_{\mathcal{N}}(q)$. Since $\mathcal{N}$ is fully-deterministic, there must exist $\phi \in \text{dom}(F_{\mathcal{N}}(p))$ such that $\phi(x)$ is defined. Also,

$$F'(p) = F_{\mathcal{N}}(p) \oplus (F(p) \rhd \{p\})$$

and it is clear that $\text{dom}(F'(p)) \supseteq \text{dom}(F_{\mathcal{N}}(p))$, so $\phi \in \text{dom}(F'(p))$.

complete w.r.t. Y: $\mathcal{M}$ is fully-deterministic, so $I' = \{q_0\}$. It is required that for any $y \in Y$ there exist a path p, with $\text{len}(p_\Phi) = \text{len}(\alpha^*(y))$, starting at q_0, such that $\beta'(|p|\,(\alpha'(y)))$ is defined, for $\mathcal{M}'$ to be complete w.r.t. Y.

Since $\mathcal{M}$ is complete w.r.t. Y, there is a path p in $\mathcal{M}$, fitting all the conditions for arbitrary $y \in Y$. Now, either q occurs in the Q-path, p_Q (of states passed through due to y), or it does not. If it does not, then p can be taken through $\mathcal{M}'$, and possesses all the required properties.

If q occurs in p_Q, then construct the path p' for y through $\mathcal{M}'$ as follows:

1. Let $\alpha(y) = (\langle\rangle, m_0, S)$.
2. Let p^1 be the longest path (through $\mathcal{M}$) from q_0 to q such that $\left|p^1\right|(\langle\rangle, m_0, S)$ is defined and

$$\operatorname{ran}(p^1_\Phi) \cap \operatorname{dom}(F(q) \rhd \{q\}) = \varnothing,$$

i.e. the shortest defined path not involving any of the "loop-back" arcs of q in $\mathcal{M}$.

3. Let $x^1 = (G^1, m^1, S^1) = \left|p^1\right|(\langle\rangle, m_0, S)$. Then, either $S^1 = \langle\rangle$ or it does not. If it does, then p^1 is a path meeting all the conditions.

   Otherwise, since $\mathcal{M}$ is complete w.r.t. Y, there is a $\phi \in \operatorname{dom}(F(q))$ such that $\phi(x_1)$ is defined. Therefore, there is also a $\phi' \in \operatorname{dom}(F_\mathcal{N}(q))$ that is defined for x^1.
4. Let p^2 be the longest path from q to q in $\mathcal{N}$ such that

$$\mathit{head}\, p^2_\Phi = \phi'$$

and

$$\operatorname{ran}(p^2_\Phi) \subseteq (\Phi_\mathcal{N} \setminus \operatorname{dom}(F(q) \rhd \{q\})),$$

and $\left|p^2\right|(x^1)$ is defined.
i.e. the longest path in $\mathcal{N}$ not involving any of the arcs that are "state-change" arcs in $\mathcal{M}$.

5. Let $x^2 = (G^2, m^2, S^2) = \left|p^2\right|(x^1)$. Then, either $S^2 = \langle\rangle$ or it does not. If it does, then $p^1 \frown p^2$ is a path meeting all the conditions.

   Otherwise, since $\mathcal{M}$ is complete w.r.t. Y, there is a $\phi \in \operatorname{dom}(F(q))$ such that $\phi(x^2)$ is defined. Furthermore, ϕ must be a "state-change" arc in $\mathcal{M}$.
6. Repeat the above steps until the input sequence is empty, $S^n = \langle\rangle$.

The complete path p' is a concatenation of the paths p^1 to p^n, and has the required properties.

□

4.4.2 General refinements.

The simple refinements described in section 4.4.1 are useful in many situations. However, in some cases, an X-machine-let with a richer data set will be needed to adequately model the refinement.

Definition 4.4.6 Suppose $\mathcal{M}$ is a fully-deterministic X-machine, with

$$\mathcal{M} = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T),$$

and $\mathcal{N}$ is a fully-deterministic X-machine-let, with

$$\mathcal{N} = (X_\mathcal{N}, Y, Z, \alpha, \beta, Q_\mathcal{N}, \Phi_\mathcal{N}, F_\mathcal{N}, \{q\}, \{q\}),$$

where $q \in Q \cap Q_\mathcal{N}$ and

$$X_\mathcal{N} \supseteq X.$$

Further, for any $x \in X$ such that $\exists\, \phi \in \operatorname{dom}(F(q))$ for which $\phi(x)$ is defined, there must be a $\phi' \in \operatorname{dom}(F_\mathcal{N}(q))$ such that $\phi'(x)$ is defined. And also, for any $x \in (X_\mathcal{N} \setminus X)$ there is no $\phi \in \operatorname{dom}(F_\mathcal{N}(q))$ such that $\phi(x)$ is defined.

Then X-machine $\mathcal{M}'$ is a *Q-general-refinement* of $\mathcal{M}$ using $\mathcal{N}$ if:

$$\mathcal{M}' = (X', Y', Z', \alpha', \beta', Q', \Phi', F', I', T'),$$

where

$$
\begin{aligned}
X' &= X_{\mathcal{N}} \\
Y' &= Y \\
Z' &= Z \\
\alpha' &= \alpha \\
\beta' &= \beta \\
Q' &= Q \cup Q_{\mathcal{N}} \\
&\quad Q \cap Q_{\mathcal{N}} = \{q\} \\
\Phi' &= \Phi \cup \Phi_{\mathcal{N}} \\
&\quad \Phi \cap \Phi_{\mathcal{N}} \supseteq \operatorname{dom}(F(q) \rhd \{q\}) \\
F' &= \lambda\, p : Q' \bullet \begin{cases} F(p) & \text{if } p \in (Q \setminus \{q\}) \\ F_{\mathcal{N}}(p) & \text{if } p \in (Q_{\mathcal{N}} \setminus \{q\}) \\ f_q & \text{if } p = q \end{cases} \\
I' &= I \\
T' &= T \\
I_{\mathcal{N}} &= q
\end{aligned}
$$

and f_q is defined as follows:

$$f_q = F_{\mathcal{N}}(q) \oplus (F(q) \rhd \{q\})$$

$\diamond$

Less formally, a general refinement of $\mathcal{M}$ using $\mathcal{N}$ to produce $\mathcal{N}'$ can be described as a simple refinement with the following additional features:

1. X is extended to X', but only the "new" ϕ's from $\mathcal{N}$ can use the extra values of X'.
2. So, for the common state q, MAttainable(q) $\subseteq$ Mem(X) in both $\mathcal{M}$ and $\mathcal{M}'$.

Theorem 4.4.7 Suppose $\mathcal{M}$ is a stream-X-machine, and $\mathcal{N}$ a stream-X-machine-let with q in common with $\mathcal{M}$, $\mathcal{M}$ and $\mathcal{N}$ are both fully-deterministic and complete w.r.t. Y, and $\mathcal{M}$ is Q-Γ-minimal.

$$
\begin{aligned}
\mathcal{M} &= (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T) \\
\mathcal{N} &= (X_{\mathcal{N}}, Y, Z, \alpha, \beta, Q_{\mathcal{N}}, \Phi_{\mathcal{N}}, F_{\mathcal{N}}, \{q\}, \{q\})
\end{aligned}
$$

Then $\mathcal{M}'$, the Q-general-refinement of $\mathcal{M}$, using $\mathcal{N}$ is a stream-X-machine that is fully-deterministic and complete w.r.t. Y.

Proof Since the extra values in X' are only used in the "new" bit of $\mathcal{M}'$, all of the transitions in the rest of the machine are unaffected. Therefore, the proof can follow exactly the same lines as that for theorem 4.4.5. $\square$

Chapter 5

Case study: Specification using X-machines.

In this chapter I introduce a model of a relatively large software system, illustrating how X-machines can be used to build and refine the specification of such a system.

5.1 The Petri-net tool.

The model is of a tool for building, manipulating and animating Petri-nets, and is based on a real implementation, carried out by Connelly [5].

Petri-nets are process models, originally developed to model population movement, but now used to model all kinds of control systems, and synchronous and asynchronous circuit. The details are not particularly important to the discussion that follows, but can be found in Reisig's introductory book, [50].

The tool allows the user to construct a net from *places* (circles), *transitions* (bars) and directed *arcs* (arrows) between them, by pointing and clicking with a mouse. *Tokens* (dots) can be put in the places, and then the net can be animated, causing the tokens to move around the diagram according to the Petri-net rules.

5.2 Brief introduction to Petri-nets.

See Reisig's book [50] for more details. There are two parts to a Petri-net, the net ($\mathcal{P}$), and the marking (μ).

Definition 5.2.1 The *net*, $\mathcal{P}$, is defined as follows

$$\mathcal{P} : \textsc{Net} == \textsc{Places} \times \textsc{Transitions} \times \textsc{InArcs} \times \textsc{OutArcs}$$

where

$$\begin{aligned}
\textsc{Places} &== \mathbb{P}\,\textsc{Names}\\
\textsc{Transitions} &== \mathbb{P}\,\textsc{Names}\\
\textsc{InArcs} &== \textsc{Transitions} \rightarrow \mathrm{bag}(\textsc{Places})\\
\textsc{OutArcs} &== \textsc{Transitions} \rightarrow \mathrm{bag}(\textsc{Places}).
\end{aligned}$$

Names is just a set of suitable names for places and transitions. These have to include those entered by users, and those generated automatically by the system.

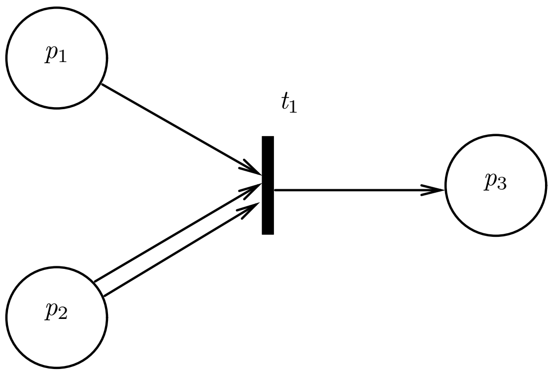


Figure 5.1: A very simple Petri-net.

Notation 5.2.2
$\mathcal{P}$.PLACES refers to the PLACES part of the Petri-net $\mathcal{P}$. Similarly, for $\mathcal{P}$.TRANSITIONS, *etc.* ◇

$\mathcal{P}$.INARCS is the function that describes arcs leading into the transitions of $\mathcal{P}$.
Similarly, $\mathcal{P}$.OUTARCS describes the arcs leading out of transitions of $\mathcal{P}$.

The following conditions must also be met:

- $\mathcal{P}.\text{PLACES} \cap \mathcal{P}.\text{TRANSITIONS} = \varnothing$.
  In other words, the places and transitions of a net have different names.
- $\text{dom}(\text{ran}(\mathcal{P}.\text{INARCS})) \subseteq \mathcal{P}.\text{PLACES}$.
  In other words, the arcs into a transition must come from places that exist in the net.
- $\text{dom}(\text{ran}(\mathcal{P}.\text{OUTARCS})) \subseteq \mathcal{P}.\text{PLACES}$.
  In other words, the arcs from a transition must go to places that exist in the net.

◇

Example 5.2.3 Consider figure 5.1. It would be represented as follows:

1. $\mathcal{P}.\text{PLACES} = \{p_1, p_2, p_3\}$
2. $\mathcal{P}.\text{TRANSITIONS} = \{t_1\}$
3. $\mathcal{P}.\text{INARCS} = \{t_1 \mapsto [\![p_1, p_2, p_2]\!]\}$
4. $\mathcal{P}.\text{OUTARCS} = \{t_1 \mapsto [\![p_3]\!]\}$

◇

Definition 5.2.4 The marking, μ, of net $\mathcal{P}$ is defined as:

$$\mu : \text{MARKING} == \text{PLACES} \twoheadrightarrow \mathbb{N}$$

with the following condition

$$\text{dom}(\mu) = \mathcal{P}.\text{PLACES}$$

In other words, every place in the net maps onto a natural number, representing the number of *tokens* at the place. ◇

Example 5.2.5 If the net from example 5.2.3 had

$$\mu = \{p_1 \mapsto 1, p_2 \mapsto 3, p_3 \mapsto 0\}$$

this would represent a marking of 1 token in p_1, 3 tokens in p_2 and no tokens in p_3. $\diamond$

Given a Petri-net (consisting of a net and a marking), there are rules for moving the tokens around the net:

Definition 5.2.6 Given a net $\mathcal{P}$ and marking μ for $\mathcal{P}$, a transition, t, is *enabled* iff:

$$\forall\, p \in \mathrm{dom}(\mathcal{P}.\textsc{InArcs}(t)) \bullet \mu(p) \geq (\mathcal{P}.\textsc{InArcs}(t))(p)$$

i.e. if there are enough tokens in all the places leading into t. $\diamond$

Example 5.2.7 The marking in example 5.2.5 means that t_1 is enabled. $\diamond$

Definition 5.2.8 If a transition t is enabled in net $\mathcal{P}$ with marking μ_1, it can *fire* to produce marking μ_2:

$$\mu_2 = \mu_1 \oplus (\mathit{RemovedTokens} \oplus \mathit{AddedTokens})$$

where *RemovedTokens* is a subfunction describing those places that have had tokens removed due to the firing of t, and *AddedTokens* is a subfunction describing those places that have had tokens added as a result of t firing.

$$\begin{aligned}
\mathit{RemovedTokens} &= \{p : \textsc{InArcs} \mid p \in \mathrm{dom}(\mathcal{P}.\textsc{InArcs}(t)) \wedge \\
&\qquad\qquad n = \mu(p) - (\mathcal{P}.\textsc{InArcs}(t))(p) \bullet p \mapsto n\} \\
\mathit{AddedTokens} &= \{p : \textsc{OutArcs} \mid p \in \mathrm{dom}(\mathcal{P}.\textsc{OutArcs}(t)) \wedge \\
&\qquad\qquad n = \mu(p) + (\mathcal{P}.\textsc{OutArcs}(t))(p) \bullet p \mapsto n\}
\end{aligned}$$

$\diamond$

5.3 The screen.

The Petri-net tool divides the screen into three areas: the menu bar, the dialogue box and the main area (see figure 5.2). The mouse driven cursor can be in any of the areas. Screen positions are elements of the type POSITIONS, which simply gives the Cartesian co-ordinates of each of the possible screen locations.

1. The menu bar consists of a line of boxes across the top of the screen, with the names of the user selectable functions and modes written in them. The box corresponding to the current mode is coloured in.

   Each box occupies an area, and the following sets of co-ordinates are defined accordingly:

   $$\begin{gathered}\mathit{SaveSpace}, \mathit{LoadSpace}, \mathit{RunSpace}, \mathit{PlaceSpace}, \mathit{TransitionSpace}, \mathit{ArcSpace}, \\ \mathit{TokenSpace}, \mathit{MoveSpace}, \mathit{NameSpace}, \mathit{ClearSpace}, \mathit{QuitSpace} : \mathbb{P}\,\textsc{Positions}\end{gathered}$$

   Also, define *MenuBar* to be the union of all these sets.

2. The dialogue box is a single line at the bottom of the screen, on which error messages appear, and the user's keyboard input is echoed.

3. The main area covers the rest of the screen, and the Petri-nets are drawn onto it.

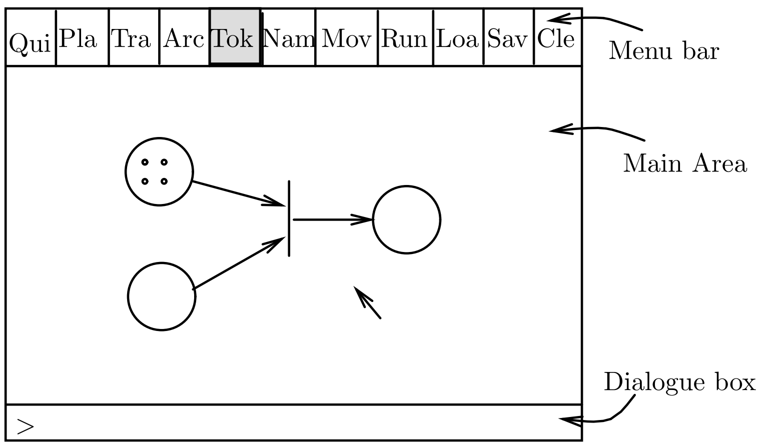


Figure 5.2: The screen layout of the Petri-net tool.

5.4 X-machine model of the tool.

The model is a stream-X-machine called $\mathcal{M}_1$, and

$$\mathcal{M}_1 = (X, Y, Z, \alpha, \beta, Q, \Phi, F, I, T).$$

5.4.1 The fundamental data set, X.

$X = \Gamma^* \times M \times \Sigma^*$, as usual for a stream-$X$-machine:

The memory (M)

M describes a Petri-net, its current marking, the positions (on the screen) of the places and transitions, and the status of the menu and dialogue box. So $(\mathcal{P}, \mu, \mathcal{L}) : M == \textsc{Net} \times \textsc{Marking} \times \textsc{Locations}$.

1. The net, $\mathcal{P}$ as in definition 5.2.1.
2. The marking of the net, μ, as in definition 5.2.4.
3. The locations of places and transitions on the screen, and the status of the menu and dialogue box ($\mathcal{L}$):

   First introduce a set to represent all screen locations,

   $$\textsc{Positions} == X \times Y.$$

   So $(x, y) \in \textsc{Positions}$ are simply the Cartesian coordinates of a screen position.

   Now, introduce the actual structure $\mathcal{L}$:

   $$\mathcal{L} : \textsc{Locations} == \pi \times \tau \times \nu \times \delta$$

   where

   $$\begin{aligned} \pi &== \textsc{Positions} \nrightarrow \textsc{Places} \\ \tau &== \textsc{Positions} \nrightarrow \textsc{Transitions} \\ \nu &== \{\textsc{Pl}, \textsc{Tr}, \textsc{Ar}, \textsc{To}, \textsc{Ru}, \textsc{Mo}, \textsc{Na}, \textsc{Lo}, \textsc{Sa}, \textsc{Cl}, \textsc{Qu}\} \\ \delta &== \{a, \ldots, z, \textit{etc.}\}^* \end{aligned}$$

   and the following conditions are met:

- $\operatorname{ran}(\mathcal{L}.\pi) = \mathcal{P}.\textsc{Places}$
- $\operatorname{ran}(\mathcal{L}.\tau) = \mathcal{P}.\textsc{Transitions}$

So, given $\mathcal{L} : \textsc{Locations}$,

(a) $\mathcal{L}.\pi$ is a partial function mapping points on the screen onto particular places. For each place in the net, there is a maplet $\{c \mapsto \mathit{place}\}$, where c(: $\textsc{Positions}$) is the centre point of the place.

(b) Similarly, $\mathcal{L}.\tau$ is a partial function mapping points on the screen onto particular transitions.

(c) $\mathcal{L}.\nu$ gives the status of the menu bar across the top of the screen, so that the corresponding space on the screen is coloured in.

(d) $\mathcal{L}.\delta$ is a sequence of letters and characters, that represents the contents of the dialogue box, along the bottom of the screen.

The input stream, Σ^*.

The input alphabet has to describe all of the possible inputs:

$\Sigma == \textsc{Positions} \times \textsc{MButtons} \times \textsc{Keyboard}$

where the $\textsc{Positions}$ element represents the mouse's position on the screen, the $\textsc{MButtons}$ element represents which mouse buttons are pressed, and the $\textsc{Keyboard}$ element represents which keyboard buttons are pressed.

$$\begin{array}{lcl}
\textsc{MButtons} & == & \{\lhd, \rhd, \diamond\} \\
\lhd & = & \text{left mouse button} \\
\rhd & = & \text{right mouse button} \\
\diamond & = & \text{no mouse button} \\
\textsc{Keyboard} & == & \{a, \ldots, z, \mathit{Return}, \oslash\} \\
\oslash & = & \text{no keyboard input}
\end{array}$$

Assume that no more than one button on the mouse, and one button on the keyboard can be pressed at a time.

The output stream, Γ^*.

The output alphabet has to describe all of the possible screen appearances. There are three parts to the screen, described in section 5.3. Also, the cursor appears on the screen. So:

$\Gamma == \textsc{Positions} \times \textsc{Locations} \times \textsc{InArcs} \times \textsc{OutArcs} \times \textsc{Marking}$,

where the $\textsc{Positions}$ element records the current cursor position, the $\textsc{Locations}$ element the positions of the places and transitions on the screen, plus the menu bar and dialogue box statuses, the $\textsc{InArcs}$ and $\textsc{OutArcs}$ elements the locations of the arcs on the screen, and the $\textsc{Marking}$ element the positions of tokens.

5.4.2 The input set, Y.

$Y == \Sigma^*$.

5.4.3 The output set, Z.

$Z == \Gamma^*$.

5.4.4 The input function, α.

$$\alpha = \lambda\, S : \Sigma^* \bullet (\langle\rangle, (\mathcal{P}_0, \mu_0, \mathcal{L}_0), S)$$

where $\mathcal{P}_0$ is an empty net, μ_0 an empty marking, and $\mathcal{L}_0$ an empty screen description.

5.4.5 The output function, β.

$\beta = \lambda(S, m, G) : X \bullet G.$

5.4.6 The states, Q.

$Q = \{\textsc{Place}, \textsc{Arc}, \textsc{Transition}, \textsc{Token}, \textsc{Run}, \textsc{Name}, \textsc{Move}, \textsc{Save}, \textsc{Load}\}$

5.4.7 The transition functions, Φ.

Firstly, I define a few useful auxiliary functions, so that the definitions proper are simpler.

$$\begin{aligned}
&\mathit{PSpace} : \mathbb{P}\,\textsc{Positions} \rightarrow \mathbb{P}\,\textsc{Positions}\\
&\qquad = \lambda\, P \bullet \{(x, y) : \textsc{Positions} \mid\\
&\qquad\qquad \exists (x', y') \in P \bullet (x - x')^2 + (y - y')^2 \leq (\mathit{PlaceRadius})^2\}\\
&\mathit{TSpace} : \mathbb{P}\,\textsc{Positions} \rightarrow \mathbb{P}\,\textsc{Positions}\\
&\qquad = \lambda\, P \bullet \{(x, y) : \textsc{Positions} \mid\\
&\qquad\qquad \exists (x', y') \in P \bullet |x - x'| \leq \mathit{TransitionWidth} \wedge\\
&\qquad\qquad\qquad |y - y'| \leq \mathit{TransitionHeight}\}
\end{aligned}$$

where *PlaceRadius* is the radius of a place on the screen, and *TransitionWidth* and *TransitionHeight* are half of the width and height respectively of a transition on the screen (a transition extends *TransitionHeight* above and below its centre point, *etc.*).

So, given a set of Positions, *PSpace* treats them as the centres of places and returns the set of Positions representing the area covered by those places on the screen. Similarly, *TSpace* returns the set of Positions representing the area covered on the screen by transitions centred on the Positions in the set passed as an argument.

Next, I define functions for locating the centre-point of places and transitions. Given an element of Positions located within a place on the screen, *PMiddle* returns its centre point. *TMiddle* behaves similarly for transitions.

$$\mathit{PMiddle} : \textsc{Positions} \times \pi \rightarrow \textsc{Positions} = \lambda(c, p) : \textsc{Positions} \times \pi \bullet$$
$$\begin{cases} \bot & \text{if } c \notin \mathit{PSpace}(\mathrm{dom}(p)) \\ & \text{ie, if } c \text{ isn't in a place at all} \\ c' & \text{if } c \in \mathit{PSpace}(\mathrm{dom}(p)) \end{cases}$$

where: $c' \in \mathrm{dom}(p)$, and
$c \in \mathit{PSpace}(\{c'\})$

$$\mathit{TMiddle} : \textsc{Positions} \times \tau \rightarrow \textsc{Positions} = \lambda(c, t) : \textsc{Positions} \times \tau \bullet$$
$$\begin{cases} \bot & \text{if } c \notin \mathit{TSpace}(\mathrm{dom}(t)) \\ & \text{ie, if } c \text{ isn't in a transition at all} \\ c' & \text{if } c \in \mathit{TSpace}(\mathrm{dom}(t)) \end{cases}$$

where: $c' \in \mathrm{dom}(t)$, and
$c \in \mathit{TSpace}(\{c'\})$

So long as none of the places or transitions overlap, *PMiddle* and *TMiddle* are well defined.

PSpace and *TSpace* also allow me to define functions for recognising when a set of POSITIONS overlaps with the places, transition or both on the screen.

$$\mathit{Overlap} = \lambda(\mathit{Points}, \mathcal{L}) : \mathbb{P}\,\textsc{Positions} \times \textsc{Locations} \bullet \begin{cases} \text{true} & \text{if } \mathit{Points} \cap \\ & (\mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \cup \mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau))) \neq \varnothing \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{PlcOverlap} = \lambda(\mathit{Points}, \mathcal{L}) : \mathbb{P}\,\textsc{Positions} \times \textsc{Locations} \bullet \begin{cases} \text{true} & \text{if } \mathit{Points} \cap (\mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \neq \varnothing \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{TransOverlap} = \lambda(\mathit{Points}, \mathcal{L}) : \mathbb{P}\,\textsc{Positions} \times \textsc{Locations} \bullet \begin{cases} \text{true} & \text{if } \mathit{Points} \cap (\mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau)) \neq \varnothing \\ \text{false} & \text{otherwise} \end{cases}$$

Given a set of POSITIONS, *Points*, and a single element of LOCATIONS, $\mathcal{L}$, *Overlap* returns true if the set *Points* overlaps with any of the places or transitions of $\mathcal{L}$. *PlcOverlap* and *TransOverlap* are similar, but are for overlaps with just places and just transitions respectively.

Finally, I define some functions to make checking for input conditions (involving the mouse and keyboard buttons) simpler:

$$\mathit{LeftPressed} = \lambda\, m : \textsc{MButtons} \bullet \begin{cases} \text{true} & \text{if } m = \triangleleft \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{RightPressed} = \lambda\, m : \textsc{MButtons} \bullet \begin{cases} \text{true} & \text{if } m = \triangleright \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{NonePressed} = \lambda\, m : \textsc{MButtons} \bullet [\neg\, \mathit{LeftPressed}(m)] \wedge [\neg\, \mathit{RightPressed}(m)]$$

$$\mathit{NoKeysPressed} = \lambda\, k : \textsc{Keyboard} \bullet \begin{cases} \text{true} & \text{if } k = \oslash \\ \text{false} & \text{otherwise} \end{cases}$$

Using these auxiliary functions, I can define each of the transition functions of $\mathcal{M}_1$ in turn. The functions all follow the same pattern, each one being defined for a restricted set of possible combinations of input and the existing status of the memory. So, after the first few definitions, I keep explanatory notes to a minimum.

The function to add a place to the net, AddPlace:

$$\mathsf{AddPlace} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), ((c, m, k) :: S)) : X \bullet$$

$$\begin{cases} ((c,\mathcal{L},i,o,\mu) :: G, \\ \quad ((P,T,i,o),\mu,\mathcal{L}),S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{Overlap}(\mathit{PSpace}\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, left button pressed too close to an existing} \\ & \text{place or transition} \\ ((c,\mathcal{L}',i,o,\mu) :: G, \\ \quad ((P',T,i,o),\mu,\mathcal{L}'),S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \neg\ \mathit{Overlap}(\mathit{PSpace}\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, left button pressed away from all the ex-} \\ & \text{isting places and transitions} \\ \perp & \text{otherwise} \end{cases}$$

where,

$$\begin{aligned} \mathcal{L}' &= (pl,\mathcal{L}.\tau,\mathcal{L}.\nu,\mathcal{L}.\delta) \\ P' &= P \cup \{\mathit{newname}\} \\ pl &= \mathcal{L}.\pi \cup \{c \mapsto \mathit{newname}\} \\ \mathit{newname} &\notin P \cup T. \end{aligned}$$

AddPlace is defined iff the left mouse button is pressed somewhere in the main area of the screen. If it is pressed with the mouse positioned close enough to the existing places and transitions that a new place would overlap them, no new place is added. On the other hand, if the mouse is pressed far enough away from the existing places and transitions, then a new place is added at that position, with a *newname* determined in some way beyond the scope of this specification.

The function to remove a place from the net, DelPlace**:**

$$\mathsf{DelPlace} = \lambda(G,((P,T,i,o),\mu,\mathcal{L}),(c,m,k) :: S) : X \bullet$$

$$\begin{cases} ((c,\mathcal{L},i,o,\mu) :: G, \\ \quad ((P,T,i,o),\mu,\mathcal{L}),S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \neg\ \mathit{PlcOverlap}(\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, right button pressed outside all the exist-} \\ & \text{ing places} \\ ((c,\mathcal{L}',i',o',\mu') :: G, \\ \quad ((P',T,i',o'),\mu',\mathcal{L}') & \text{if } \mathit{RightPressed}(m) \wedge \\ & \mathit{PlcOverlap}(\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, right button pressed inside an existing} \\ & \text{place} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathcal{L}' &= (pl,\mathcal{L}.\tau,\mathcal{L}.\nu,\mathcal{L}.\delta) \\ P' &= P \setminus \{p\} \\ i' &= \lambda\, t : \textsc{Transitions} \bullet (\{p\} \lhd i(t)) \\ o' &= \lambda\, t : \textsc{Transitions} \bullet (\{p\} \lhd o(t)) \\ \mu' &= P' \lhd \mu \\ pl &= \mathcal{L}.\pi \rhd P' \\ p &= P_L(\mathit{PMiddle}(c)) \end{aligned}$$

DelPlace is defined iff the right mouse button is pressed somewhere in the main area of the screen. If it is pressed inside an existing place on the screen, then that place is removed from the net, along with its tokens, and all the arc leading to or from it. If it is pressed outside all the existing places, nothing happens.

The function to add a transition, AddTransition:

$$\mathsf{AddTransition} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, & \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{Overlap}(\mathit{TSpace}\{c\}, \mathcal{L}) \\ & \textit{i.e}\text{, left button pressed too close to an existing} \\ & \text{place or transition} \\ ((c, \mathcal{L}', i, o, \mu) :: G, & \\ \quad ((P, T', i, o), \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \neg\ \mathit{Overlap}(\mathit{TSpace}\{c\}, \mathcal{L}) \\ & \textit{i.e}\text{, left button pressed away from all the ex-} \\ & \text{isting places and transitions} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathcal{L}' &= (\mathcal{L}.\pi, \mathit{tr}, \mathcal{L}.\nu, \mathcal{L}.\delta) \\ T' &= T \cup \{\mathit{newname}\} \\ \mathit{tr} &= \mathcal{L}.\tau \cup \{c \mapsto \mathit{newname}\} \\ \mathit{newname} &\notin P \cup T. \end{aligned}$$

The function to remove a transition, DelTransition:

$$\mathsf{DelTransition} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, & \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \neg\ \mathit{TransOverlap}(\{c\}, \mathcal{L}) \\ & \textit{i.e}\text{, right button pressed away from all the ex-} \\ & \text{isting transitions} \\ ((c, \mathcal{L}', i', o', \mu) :: G, & \\ \quad ((P, T', i', o'), \mu, \mathcal{L}'), S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \mathit{TransOverlap}(\{c\}, \mathcal{L}) \\ & \textit{i.e}\text{, right button pressed inside an existing} \\ & \text{transition} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} t &= T_L(\mathit{TMiddle}(c)) \\ T' &= T \setminus \{t\} \\ \mathit{tr} &= \mathcal{L}.\tau \rhd T' \\ \mathcal{L}' &= (\mathcal{L}.\pi, \mathit{tr}, \mathcal{L}.\nu, \mathcal{L}.\delta) \\ i' &= \{t\} \lhd i \\ o' &= \{t\} \lhd o \end{aligned}$$

The function to put a token into a place, $\mathsf{AddToken}$**:**

$$\mathsf{AddToken} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \neg\ \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ & i.e\text{, left button pressed outside all the existing} \\ & \text{places} \\ ((c, \mathcal{L}, i, o, \mu') :: G, \\ \quad ((P, T, i, o), \mu', \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ & i.e\text{, left button pressed inside an existing place} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mu' &= \mu \oplus \{\mathit{place} \mapsto (\mu(\mathit{place}) + 1)\} \\ \mathit{place} &= P_L(\mathit{PMiddle}(c)) \end{aligned}$$

The function to remove a token, $\mathsf{DelToken}$**:**

$$\mathsf{DelToken} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, \\ \quad ((P, T, i, o), \mu, \mathcal{L}, S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \neg\ \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ & i.e\text{, right button pressed outside all the exist-} \\ & \text{ing places} \\ ((c, \mathcal{L}, i, o, \mu') :: G, \\ \quad ((P, T, i, o), \mu', \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ & i.e\text{, right button pressed inside an existing} \\ & \text{place} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mu' &= \mu \oplus \{\mathit{place} \mapsto n\} \\ \mathit{place} &= P_L(\mathit{PMiddle}(c)) \\ n &= \begin{cases} 0 & \text{if } \mu(\mathit{place}) \leq 1 \\ \mu(\mathit{place}) - 1 & \text{otherwise} \end{cases} \end{aligned}$$

The function to change the name of a place or transition, $\mathsf{ChangeName}$**:**

$$\mathsf{ChangeName} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$

$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, & \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \neg\, \mathit{Overlap}(\{c\}, \mathcal{L}) \\ & i.e, \text{ left button pressed outside any of the ex-} \\ & \text{isting places or transitions.} \\ ((c, \mathcal{L}^P, i', o', \mu') :: G, & \\ \quad ((P', T, i', o'), \mu', \mathcal{L}^P), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ & i.e, \text{ left button pressed in an existing place} \\ ((c, \mathcal{L}^T, i', o', \mu) :: G, & \\ \quad ((P, T', i', o'), \mu, \mathcal{L}^T), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{TransOverlap}(\{c\}, \mathcal{L}) \\ & i.e, \text{ left button pressed in an existing trans-} \\ & \text{ition} \\ \bot & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}^P = (pl, \mathcal{L}.\tau, \mathcal{L}.\nu, \langle newname \rangle)$$
$$\mathcal{L}^T = (\mathcal{L}.\pi, tr, \mathcal{L}.\nu, \langle newname \rangle)$$

$$name = \begin{cases} P_L(\mathit{PMiddle}(c)) & \text{if } \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ T_L(\mathit{TMiddle}(c)) & \text{if } \mathit{TransOverlap}(\{c\}, \mathcal{L}) \end{cases}$$
$$newname \notin (P \cup T) \setminus \{name\}$$

$$pl = \lambda\, c : \textsc{Positions} \bullet \begin{cases} (\mathcal{L}.\pi)(c) & \text{if } (\mathcal{L}.\pi)(c) \neq name \\ newname & \text{if } \mathcal{L}.\pi(c) = name \end{cases}$$
$$P' = (P \setminus \{name\}) \cup \{newname\}$$

$$tr = \lambda\, c : \textsc{Positions} \bullet \begin{cases} (\mathcal{L}.\tau)(c) & \text{if } (\mathcal{L}.\tau)(c) \neq name \\ newname & \text{if } (\mathcal{L}.\tau)(c) = name \end{cases}$$
$$T' = (T \setminus \{name\}) \cup \{newname\}$$

$$i' = \begin{cases} \lambda\, t : \textsc{Transitions} \bullet (\{name\} \lhd i(t)) \oplus & \\ \quad \{newname \mapsto (i(t))(name)\} & \text{if } name \in P \\ (\{name\} \lhd i) \oplus \{newname \mapsto i(name)\} & \text{if } name \in T \end{cases}$$
$$o' = \begin{cases} \lambda\, t : \textsc{Transitions} \bullet (\{name\} \lhd o(t)) \oplus & \\ \quad \{newname \mapsto (o(t))(name)\} & \text{if } name \in P \\ (\{name\} \lhd o) \oplus \{newname \mapsto o(name)\} & \text{if } name \in T \end{cases}$$

$$\mu' = (\{name\} \lhd \mu) \oplus \{newname \mapsto \mu(name)\}$$

ChangeName doesn't attempt to deal with the way in which a name is typed in by the user. It just assumes that *newname* appears, "by magic", and that it is different from any of the existing names.

The function to display a name, ShowName:

$$\mathsf{ShowName} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$

$$\begin{cases} ((c,\mathcal{L},i,o,\mu) :: G \\ \quad ((P,T,i,o),\mu,\mathcal{L}),S) & \text{if } \mathit{NonePressed}(m) \wedge \\ & \neg\ \mathit{Overlap}(\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, the mouse is pointed away from all the} \\ & \text{existing places and transitions} \\ ((c,\mathcal{L}',i,o,\mu,\langle \mathit{name}\rangle) :: G \\ \quad ((P,T,i,o),\mu,\mathcal{L}'),S) & \text{if } \mathit{NonePressed}(m) \wedge \\ & \mathit{Overlap}(\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, the mouse is pointed at an existing place} \\ & \text{or transition} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathit{name} &= \begin{cases} P_L(\mathit{PMiddle}(c)) & \text{if } \mathit{PlcOverlap}(\{c\},P_L,T_L) \\ T_L(\mathit{TMiddle}(c)) & \text{if } \mathit{TransOverlap}(\{c\},P_L,T_L) \end{cases} \\ \mathcal{L}' &= (\mathcal{L}.\pi,\mathcal{L}.\tau,\mathcal{L}.\nu,\langle \mathit{name}\rangle \end{aligned}$$

The function to add an arc, AddArc:

For the time being, assume that the function obtains a second co ordinate, c', for the end of the arc "by magic".

$$\mathsf{AddArc} = \lambda(G,((P,T,i,o),\mu,\mathcal{L}),(c,m,k) :: S) : X \bullet$$

$$\begin{cases} ((c,\mathcal{L},i,o,\mu) :: G, \\ \quad ((P,T,i,o),\mu,\mathcal{L}),S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & (\neg\ \mathit{Overlap}(\{c\},\mathcal{L}) \vee \neg\ \mathit{Overlap}(\{c'\},\mathcal{L}) \vee \\ & (\mathit{PlcOverlap}(\{c\},\mathcal{L}) \wedge \mathit{PlcOverlap}(\{c'\}),\mathcal{L}) \vee \\ & (\mathit{TransOverlap}(\{c'\},\mathcal{L}) \wedge \\ & \mathit{TransOverlap}(\{c\},\mathcal{L}))) \\ & \textit{i.e}\text{, if one or other (or both) the positions are} \\ & \text{away from all the existing places and trans-} \\ & \text{itions, or if both the positions are in places, or} \\ & \text{if both the positions are in transitions} \\ ((c,\mathcal{L},i',o,\mu) :: G, \\ \quad ((P,T,i',o),\mu,\mathcal{L}),S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{PlcOverlap}(\{c\},\mathcal{L}) \wedge \mathit{TransOverlap}(\{c'\},\mathcal{L}) \\ & \textit{i.e}\text{, it is an I}\textsc{n}\text{A}\textsc{rc}\text{, starting at an existing place} \\ & \text{and ending at an existing transition.} \\ ((c,\mathcal{L},i,o',\mu) :: G, \\ \quad ((P,T,i,o'),\mu,\mathcal{L}),S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{TransOverlap}(\{c\},\mathcal{L}) \wedge \mathit{PlcOverlap}(\{c'\},\mathcal{L}) \\ & \textit{i.e}\text{, it is an O}\textsc{ut}\text{A}\textsc{rc}\text{, starting at an existing} \\ & \text{transition and ending at an existing place.} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} i' &= i \oplus \{t \mapsto (i(t) \oplus \{p \mapsto (i(t))(p)+1\})\} \\ t &= \mathcal{L}.\tau(\mathit{TMiddle}(c')) \\ p &= \mathcal{L}.\pi(\mathit{PMiddle}(c)) \end{aligned}$$

$$\begin{aligned} o' &= o \oplus \{t \mapsto (o(t) \oplus \{p \mapsto (o(t))(p) + 1\})\} \\ t &= \mathcal{L}.\tau(\mathit{TMiddle}(c)) \\ p &= \mathcal{L}.\pi(\mathit{PMiddle}(c')) \end{aligned}$$

The function for removing an arc, DelArc**:**

As in the definition for AddArc, I assume that a second co-ordinate, c', appears "by magic", for the time being.

$$\mathsf{DelArc} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$

$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & (\neg\ \mathit{Overlap}(\{c\}, \mathcal{L}) \vee \neg\ \mathit{Overlap}(\{c'\}, \mathcal{L}) \vee \\ & (\mathit{PlcOverlap}(\{c\}, \mathcal{L}) \wedge \mathit{PlcOverlap}(\{c'\}), \mathcal{L}) \vee \\ & (\mathit{TransOverlap}(\{c'\}, \mathcal{L}) \wedge \\ & \mathit{TransOverlap}(\{c\}, \mathcal{L}))) \\ & i.e\text{, if one or other (or both) the positions are} \\ & \text{away from all the existing places and trans-} \\ & \text{itions, or if both the positions are in places, or} \\ & \text{if both the positions are in transitions} \\ ((c, \mathcal{L}, i', o, \mu) :: G, \\ \quad ((P, T, i', o), \mu, \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \wedge \mathit{TransOverlap}(\{c'\}, \mathcal{L}) \\ & i.e\text{, it is an } \textsc{InArc}\text{, starting at an existing place} \\ & \text{and ending at an existing transition.} \\ ((c, \mathcal{L}, i, o', \mu) :: G, \\ \quad ((P, T, i, o'), \mu, \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \mathit{TransOverlap}(\{c\}, \mathcal{L}) \wedge \mathit{PlcOverlap}(\{c'\}, \mathcal{L}) \\ & i.e\text{, it is an } \textsc{OutArc}\text{, starting at an existing} \\ & \text{transition and ending at an existing place.} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} i' &= i \oplus \{t \mapsto (i(t) \oplus \{p \mapsto n\})\} \\ t &= \mathcal{L}.\tau(\mathit{TMiddle}(c')) \\ p &= \mathcal{L}.\pi(\mathit{PMiddle}(c)) \\ n &= \begin{cases} 0 & \text{if } (i(t))(p) \leq 1 \\ (i(t))(p) - 1 & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} o' &= o \oplus \{t \mapsto (o(t) \oplus \{p \mapsto n\})\} \\ t &= \mathcal{L}.\tau(\mathit{TMiddle}(c)) \\ p &= \mathcal{L}.\pi(\mathit{PMiddle}(c')) \\ n &= \begin{cases} 0 & \text{if } (o(t))(p) \leq 1 \\ (o(t))(p) - 1 & \text{otherwise} \end{cases} \end{aligned}$$

Function for moving an existing place or transition, Move

As before, I assume that the second co-ordinate required, c', appears "by magic".

$$\mathsf{Move} = \lambda(G, ((P,T,i,o),\mu,\mathcal{L}),(c,m,k) :: S) : X \bullet$$

$$\begin{cases}
\begin{array}{l}((c,\mathcal{L},i,o,\mu) :: G,\\ \quad ((P,T,i,o),\mu,\mathcal{L}),S)\end{array} & \begin{array}{l}\text{if } \mathit{LeftPressed}(m) \wedge \\ (\neg\ \mathit{Overlap}(\{c\},\mathcal{L}) \vee \\ \mathit{Overlap}(\mathit{space},p,t)) \\ \textit{i.e}\text{, if the left button is pressed and either the} \\ \text{first co-ordinate is not inside a place or trans-} \\ \text{ition, or the second co-ordinate is too close to} \\ \text{existing places or transitions}\end{array} \\[2ex]
\begin{array}{l}((c,\mathcal{L}^P,i,o,\mu) :: G,\\ \quad ((P,T,i,o),\mu,\mathcal{L}^P),S)\end{array} & \begin{array}{l}\text{if } \mathit{LeftPressed}(m) \wedge \\ \mathit{PlcOverlap}(\{c\},\mathcal{L}^P) \wedge \\ \neg\ \mathit{Overlap}(\{c'\},(p,\mathcal{L}.\tau,\mathcal{L}.\nu,\mathcal{L}.\delta) \\ \textit{i.e}\text{, if the left button is pressed and the first} \\ \text{co-ordinate is inside an existing place, and the} \\ \text{second co-ordinate is away from all the ex-} \\ \text{isting places (except for the one the first co-} \\ \text{ordinate is inside), or transitions}\end{array} \\[2ex]
\begin{array}{l}((c,\mathcal{L}^T,i,o,\mu) :: G,\\ \quad ((P,T,i,o),\mu,\mathcal{L}^T),S)\end{array} & \begin{array}{l}\text{if } \mathit{LeftPressed}(m) \wedge \\ \mathit{TransOverlap}(\{c\},\mathcal{L}) \wedge \\ \neg\ \mathit{Overlap}(\{c'\},(\mathcal{L}.\pi,t,\mathcal{L}.\nu,\mathcal{L}.\delta)) \\ \textit{i.e}\text{, if the left button is pressed, and the first} \\ \text{co-ordinate is inside an existing transition, and} \\ \text{the second co-ordinate is away from all the ex-} \\ \text{isting places and transitions (apart from the} \\ \text{one the first co-ordinate is inside}\end{array} \\
\perp & \text{otherwise}
\end{cases}$$

where

$$\begin{aligned}
\mathcal{L}^P &= (pl,\mathcal{L}.\tau,\mathcal{L}.\nu,\mathcal{L}.\delta) \\
pl &= p \oplus \{c' \mapsto \mathcal{L}.\pi(c)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}^T &= (\mathcal{L}.\pi, tr,\mathcal{L}.\nu,\mathcal{L}.\mathit{Dialogue}) \\
tr &= t \oplus \{c' \mapsto \mathcal{L}.\tau(c)\}
\end{aligned}$$

$$\mathit{space} = \begin{cases} \mathit{PSpace}(\mathit{PMiddle}(c')) & \text{if } \mathit{PlcOverlap}(\{c\},\mathcal{L}) \\ \mathit{TSpace}(\mathit{TMiddle}(c')) & \text{if } \mathit{TransOverlap}(\{c\},\mathcal{L}) \end{cases}$$

$$p = \begin{cases} \{\mathit{PMiddle}(c)\} \lhd \mathcal{L}.\pi & \text{if } \mathit{PlcOverlap}(\{c\},\mathcal{L}) \\ \mathcal{L}.\pi & \text{if } \mathit{TransOverlap}(\{c\},\mathcal{L}) \end{cases}$$

$$t = \begin{cases} \mathcal{L}.\tau & \text{if } \mathit{PlcOverlap}(\{c\},\mathcal{L}) \\ \{\mathit{TMiddle}(c)\} \lhd \mathcal{L}.\tau & \text{if } \mathit{TransOverlap}(\{c\},\mathcal{L}) \end{cases}$$

The function to save the net, Save**:**

I have not modelled the file system, so this function simply leaves the data state mostly alone.

$$\mathsf{Save} = \lambda(G, ((P,T,i,o),\mu,\mathcal{L}),(c,m,k) :: S) : X \bullet$$

$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{SaveSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

The function to load a different net, Load:

As for Save, there is no model of the file system, so the new data state appears, "by magic".

$$\mathsf{Load} = \lambda(G, M, (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{LoadSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where the new net $= ((P, T, i, o), \mu, \mathcal{L})$

The function to fire a transition, FireTransition:

The definition of FireTransition will be much clearer if a few auxiliary functions are defined first:

$$\mathit{EnabledTransitions} : \textsc{Net} \times \textsc{Marking} \rightarrow \textsc{Transitions} = \lambda(\mathcal{P}, \mu) \bullet$$
$$\{t : \mathcal{P}.\textsc{Transitions} \mid (\forall\, p \in \mathrm{dom}(\mathcal{P}.\textsc{InArcs}(t)) \bullet \mu(p) \geq (\mathcal{P}.\textsc{InArcs}(t))(p))\}$$

EnabledTransitions simply returns the set of transitions that are enabled by the given marking.

$$\mathit{Fire} : \textsc{Net} \times \textsc{Marking} \times \textsc{Names} \rightarrow \textsc{Marking} = \lambda(\mathcal{P}, \mu, t) \bullet$$
$$\begin{cases} \mu & \text{if } t \notin \mathit{EnabledTransitions}(\mathcal{P}, \mu) \\ \\ \mu \oplus (\mathit{RemovedTokens} \cup \mathit{AddedTokens}) & \text{if } t \in \mathit{EnabledTransitions}(\mathcal{P}, \mu) \end{cases}$$

where

$$\begin{aligned} \mathit{RemovedTokens} &= \{p : \mathrm{dom}(\mathcal{P}.\textsc{InArcs}(t)) \mid \\ &\qquad n = \mu(p) - (\mathcal{P}.\textsc{InArcs}(t))(p) \bullet p \mapsto n\} \\ \\ \mathit{AddedTokens} &= \{p : \mathrm{dom}(\mathcal{P}.\textsc{OutArcs}(t)) \mid \\ &\qquad n = \mu(p) + (\mathcal{P}.\textsc{OutArcs}(t))(p) \bullet p \mapsto n\}) \end{aligned}$$

Fire returns the marking of the net after the given transition has fired.

These functions now allow the main function to be defined relatively easily:

$$\mathsf{FireTransition} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$

$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \neg\ \mathit{TransOverlap}(\{c\}, \mathcal{L}) \\ & \textit{i.e}\text{, if the left button is pressed away from all} \\ & \text{the transitions} \\ ((c, \mathcal{L}, i, o, \mu') :: G, \\ \quad ((P, T, i, o), \mu', \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{TransOverlap}(\{c\}, \mathcal{L}) \\ & \textit{i.e}\text{, if the left button is pressed inside a trans-} \\ & \text{ition} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mu' = \mathit{Fire}((P, T, i, o), \mu, \mathcal{L}.\tau(\mathit{TMiddle}(c)))$$

The function to clear the system, Clear**:**

$$\mathsf{Clear} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}_0, \mathcal{P}_0.\textsc{InArcs}, \\ \qquad \mathcal{P}_0.\textsc{OutArcs}, \mu_0) :: G, \\ (\mathcal{P}_0, \mu_0, \mathcal{L}_0), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{ClearSpace} \\ & \textit{i.e.}\text{ if the button is pressed in the “Clear” area} \\ & \text{of the menu bar} \\ \perp & \text{otherwise} \end{cases}$$

The function to change to place addition/removal mode, $\mathsf{PlaceMode}$**:**

$$\mathsf{PlaceMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{PlaceSpace} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Pl}, \mathcal{L}.\delta)$$

The function to change to arc addition/removal mode, $\mathsf{ArcMode}$**:**

$$\mathsf{ArcMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{ArcSpace} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Ar}, \mathcal{L}.\delta)$$

The function to change to transition addition/removal mode, TransitionMode**:**

$$\mathsf{TransitionMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{TransitionSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Tr}, \mathcal{L}.\delta)$$

The function to change to name mode, NameMode**:**

$$\mathsf{MoveMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{NameSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Na}, \mathcal{L}.\delta)$$

The function to change to move mode, MoveMode**:**

$$\mathsf{MoveMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{MoveSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Mo}, \mathcal{L}.\delta)$$

The function to change to token addition/removal mode, TokenMode**:**

$$\mathsf{TokenMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{TokenSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{To}, \mathcal{L}.\delta)$$

The function to change to run mode, $\mathsf{RunMode}$**:**

$$\mathsf{RunMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \quad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \land c \in \mathit{RunSpace} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Ru}, \mathcal{L}.\delta)$$

The function to change to load mode, $\mathsf{LoadMode}$**:**

$$\mathsf{LoadMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} \mathsf{Load}(G, (\mathcal{P}, \mu, \mathcal{L}'), (c, m, k) :: S) & \text{if } \mathit{LeftPressed}(m) \land c \in \mathit{LoadSpace} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Lo}, \mathcal{L}.\delta)$$

$\mathsf{LoadMode}$ uses the previously defined function Load to immediately load a net in. The system then stays in the load mode, until it is switched to a new mode.

The function to change to save mode, $\mathsf{SaveMode}$**:**

$$\mathsf{SaveMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} \mathsf{Save}(G, (\mathcal{P}, \mu, \mathcal{L}'), (c, m, k) :: S) & \text{if } \mathit{LeftPressed}(m) \land c \in \mathit{SaveSpace} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Sa}, \mathcal{L}.\delta)$$

$\mathsf{SaveMode}$ uses Save to immediately save the net, and then stay in the save mode.

The function to ignore keyboard input, $\mathsf{IgnoreKeyboard}$**:**

$$\mathsf{IgnoreKeyboard} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, \\ (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{NonePressed}(m) \land \neg\ \mathit{NoKeysPressed}(k) \\ \perp & \text{otherwise} \end{cases}$$

$\mathsf{IgnoreKeyboard}$ is intended to be used when keyboard input is irrelevant. There is a check that no mouse buttons are pressed, as these should generally not be ignored.

The function to ignore the left mouse button, $\mathsf{IgnoreLeftButton}$**:**

$$\mathsf{IgnoreLeftButton} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \neg\ \mathit{Overlap}(\{c\}, \mathcal{L}) \wedge c \notin \mathit{MenuBar} \\ \bot & \text{otherwise} \end{cases}$$

The function to ignore the right mouse button, $\mathsf{IgnoreRightButton}$**:**

$$\mathsf{IgnoreRightButton} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \wedge \neg\ \mathit{Overlap}(\{c\}, \mathcal{L}) \\ \bot & \text{otherwise} \end{cases}$$

IgnoreLeftButton and IgnoreRightButton are used when left and right mouse button input respectively is irrelevant. If there were any simultaneous keyboard input, it would also be ignored. The left button is ignored if it is away from all the places and transitions, and away from the menu bar. The right button is ignored if it is away from all the places and transitions.

There are a few cases when the mouse buttons have to be more comprehensively ignored, and functions for these cases are needed.

The function to completely ignore the left mouse button, $\mathsf{CompIgnoreLeftButton}$**:**

$$\mathsf{CompIgnoreLeftButton} = \mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \notin \mathit{MenuBar} \\ \bot & \text{otherwise} \end{cases}$$

The function to completely ignore the right mouse button, $\mathsf{CompIgnoreRightButton}$**:**

$$\mathsf{CompIgnoreRightButton} = \mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \\ \bot & \text{otherwise} \end{cases}$$

The function to ignore un-interesting mouse input, $\mathsf{IgnoreMouse}$**:**

$$\mathsf{IgnoreMouse} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{NonePressed}(m) \wedge \mathit{NoKeysPressed}(k) \wedge \\ & \neg\ \mathit{Overlap}(\{c\}, \mathcal{L}) \\ \bot & \text{otherwise} \end{cases}$$

5.4.8 The state transition function, F.

There are a very large number of transitions; see figure 5.3 for the basic structure involved. The figure is cluttered enough, without adding function names to every arc. Instead, I will now describe which functions apply to which arcs.

"Mode-change" arcs.

Notice that there is an arc from each state to every other state. These arcs are the "mode change" functions. For instance,

$$\textsc{Place} \xrightarrow{\mathsf{TransitionMode}} \textsc{Transition},$$

i.e.

$$F(\textsc{Place}, \mathsf{TransitionMode}) = \textsc{Transition}.$$

Also, every state is connected to Place by Clear, *i.e*,

$$\forall\, q \in Q \bullet F(q, \mathsf{Clear}) = \textsc{Place}.$$

"Loop-back" arcs.

The remaining arcs are loop-back arcs, in that they are of the form $q \xrightarrow{\phi} q$ for state q, and function ϕ. There are several such arcs for each state, and they are listed below. After each function is listed, the values of X for which it is defined are listed, to help in checking that there is a function defined for every input in all the states (*i.e.* that the machine is complete w.r.t. Y).

Place:

AddPlace is used to add places. It is defined as long as the left mouse button is pressed somewhere on the main part of the screen.

DelPlace is used to delete places. It is defined so long as the right mouse button is pressed somewhere on the main part of the screen.

PlaceMode is used (and is only defined) if the left mouse button is pressed in *PlaceSpace*.

Clear is used (and is only defined) if the left mouse button is pressed when the mouse is pointing in *ClearSpace*.

IgnoreKeyboard is used to ignore any keyboard input. It is defined if any keyboard key is pressed, but no mouse key.

IgnoreMouse is used to ignore the mouse if no button is pressed. It is defined if the mouse is pointing anywhere, but no mouse buttons or keyboard keys are pressed.

Transition:

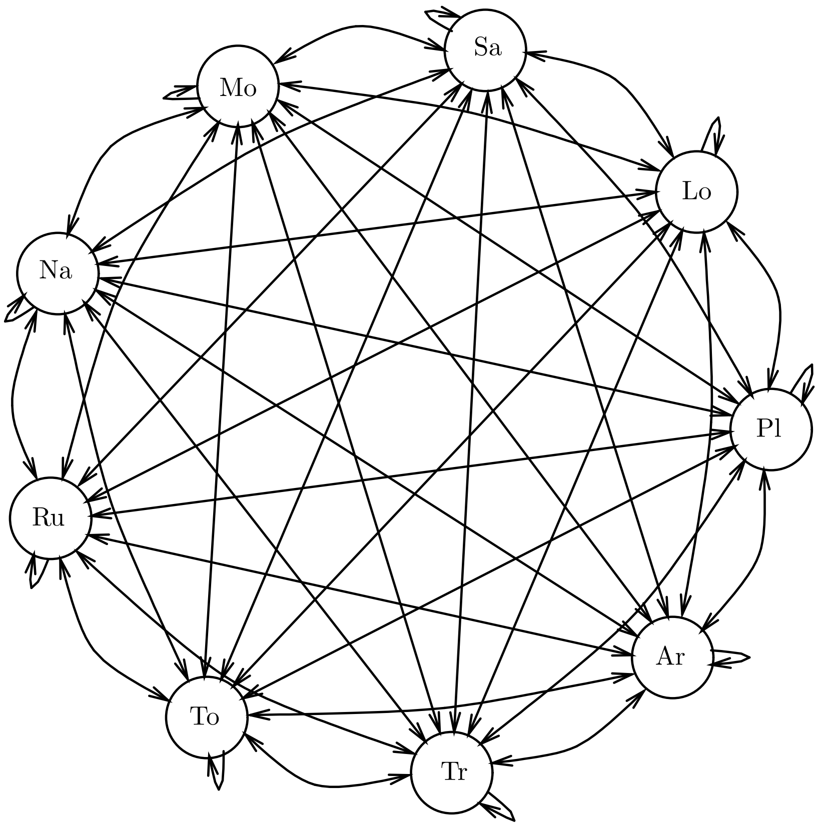
AddTransition is used to add transitions. It is defined if the left mouse button is pressed anywhere in the main area of the screen.

DelTransition is used to remove transitions. It is defined if the right mouse button is pressed, and the mouse is pointing anywhere in the main area of the screen.

TransitionMode is used (and is only defined) if the left mouse button is pressed in *TransitionSpace*.

IgnoreKeyboard is used to ignore keyboard input, where-ever the mouse is, so long as none of the mouse buttons are pressed.

IgnoreMouse is used to ignore the mouse if no button is pressed.



Pl	=	PLACE	Ar	=	ARC
Tr	=	TRANSITION	To	=	TOKEN
Ru	=	RUN	Na	=	NAME
Lo	=	LOAD	Sa	=	SAVE
Mo	=	MOVE			

In attempt to maintain clarity, no more than one arrow is drawn between each pair of states. This does not necessarily mean that there is only one transition function per arrow.
An arrow $q_1 \longleftrightarrow q_2$ indicates that there are transition functions in both directions between q_1 and q_2.

Figure 5.3: State transition diagram for the Petri-net tool

TOKEN:

AddToken is used to add tokens to places, and is defined if the left mouse button is pressed anywhere in the main area of the screen.

DelToken is used to remove tokens from places, and is defined if the right mouse button is pressed anywhere in the main area of the screen.

TokenMode is used (and only defined) if the left mouse button is pressed in *TokenSpace*.

IgnoreKeyboard is used to ignore keyboard input.

IgnoreMouse is used to ignore the mouse if no buttons are pressed.

NAME:

ChangeName is used to change the names of places and transitions. It is defined if the left mouse button is pressed anywhere in the main area of the screen.

ShowName is used to to display the name of the last place or transition that the mouse was pointing at. It is defined if the mouse is pointing anywhere in the main area of the screen, without any buttons being pressed.

NameMode is used (and only defined) if the left mouse button is pressed in *NameSpace*.

IgnoreKeyboard is used to ignore keyboard input.

CompIgnoreRightButton is used to ignore the right mouse button being pressed where-ever it is pointing.

ARC:

AddArc is used to add arcs between places and transitions, or transitions and places. It is defined if the left mouse button is pressed anywhere in the main area of the screen.

DelArc is used to remove arcs. It is defined if the right mouse button is pressed anywhere in the main area of the screen.

ArcMode is used (and only defined) if the left mouse button is pressed while the mouse is pointing in *ArcSpace*.

IgnoreKeyboard is used to ignore keyboard input.

IgnoreMouse is used to ignore the mouse if it is pointing anywhere, without any of its buttons being pressed.

MOVE:

Move is to move places or transitions around the main area of the screen. It is defined if the left mouse button is pressed anywhere in the screen.

MoveMode is used (and only defined) if the left mouse button is pressed in *MoveSpace*.

IgnoreKeyboard is used to ignore keyboard input.

CompIgnoreRightButton is used to ignore the right mouse button being pressed, where-ever it is pointing.

IgnoreMouse is used to ignore the mouse if it is pointing anywhere, without any of its buttons being pressed.

SAVE:

Save is used if the left mouse button is pressed in *SaveSpace*.

IgnoreKeyboard is used to ignore keyboard input.

CompIgnoreLeftButton is used to ignore the left mouse button being pressed, so long as the mouse is not pointing at the menu bar.

CompIgnoreRightButton is used to completely ignore the right mouse button.

IgnoreMouse is used to ignore the mouse if it is pointing anywhere, without any of its buttons being pressed.

LOAD:

Load is used if the left mouse button is pressed in *LoadSpace*.

IgnoreKeyboard is used to ignore keyboard input.

CompIgnoreLeftButton is used to ignore the left mouse button being pressed, so long as the mouse is not pointing at the menu bar.

CompIgnoreRightButton is used to completely ignore the right mouse button.

IgnoreMouse is used to ignore the mouse if it is pointing anywhere, without any of its buttons being pressed.

RUN:

FireTransition is used to fire a transition. It is defined if the left mouse button is pressed anywhere in the main area of the screen.

RunMode is used if the left mouse button is pressed in *RunSpace*.

IgnoreKeyboard is used to ignore keyboard input.

CompIgnoreRightButton is used to ignore the right mouse button, where-ever it is pressed.

IgnoreMouse is used to ignore the mouse if it is pointing anywhere, without any of its buttons being pressed.

5.4.9 The initial states, I.

$I = \{\textsc{Place}\}$.

5.4.10 The terminal states, T.

$T = Q$.

5.4.11 Quitting the machine.

The menu bar of the system includes an area marked quit. In the real system, this would be used to exit the tool. I have made no attempt to model this aspect of the system—it would require an extra arc from each state leading to a state END. This state would then have arcs corresponding to the tool closing down, and control returning to the underlying operating system, all of which are beyond the scope of this case study.

5.4.12 Operation of the model.

It is worth taking a brief look at the operation of the machine from a higher level.

The idea is that the input stream will be made up of regular "samples" of the actual input devices, taken several times a second. So, most of the input will be repetitions of relatively un-interesting inputs corresponding to the mouse pointing at a single point on the screen, with no buttons being pressed. It follows, that most of the time, the various "ignore" functions (*e.g.* IgnoreMouse) are being used, while the complex "action" functions are used relatively rarely.

Suppose $S : \Sigma^*$ is an input stream, typical real values will be along the lines of the following:

$$S = \langle \ldots, (\diamond, c, \oslash), (\diamond, c, \oslash), \ldots \rangle$$
$$S = \langle \ldots, (\diamond, c_1, \oslash), (\diamond, c_2, \oslash), (\diamond, c_3, \oslash), \ldots \rangle$$

where $c_1, c_2, c_3, \ldots$ represent a series of mouse positions, corresponding to the mouse being moved across the screen.

5.5 Discussion

The long list in section 5.4 described an X-machine model for a (somewhat simplified) version of the Petri-net tool. It presents a detailed description of each of these elements.

5.5.1 Views of the model.

At first, the complete list seems rather overwhelming in its complexity. However, one of the benefits of the X-machine model is that there are a number of simpler ways to view the system, which hide various of its complexities.

Finite-state-machine view. Treat the X-machine as a simple finite-state-machine. In other words, ignore all of the details of X, and just consider the way the machine moves from state to state; which elements of Φ connect the states together. The headings from section 5.4.7 plus section 5.4.8 provide enough information for this view.

Control view. Consider the conditions on each of the $\phi \in \Phi$, and how they are used to manage the flow of control from state to state, rather than the details of how each ϕ manipulates X. Since all of the conditions on the $\phi \in \Phi$ are input conditions, this view shows how the user's input controls the tool.

Data view. Consider the data model in X together with the manipulative parts of the $\phi \in \Phi$ in isolation (from the condition parts of the ϕ).

5.5.2 Limitations of the model.

In the course of the description of Φ (section 5.4.7), the phrase "by magic" was used a number of times. In every case some relatively sophisticated interaction needed to be handled (*e.g.* typing in the new new name for a place, using ChangeName), requiring that the user's input be stored. This raises several issues:

Firstly, it is tempting to think that the output stream, Γ^*, could be used to keep a record of past keyboard and mouse inputs, or that you could look beyond the head of the input stream, Σ^*, to find out what the next input(s) are. However, both of these possibilities are specifically ruled out by definition 4.2.3 of stream-X-machines in chapter 4.

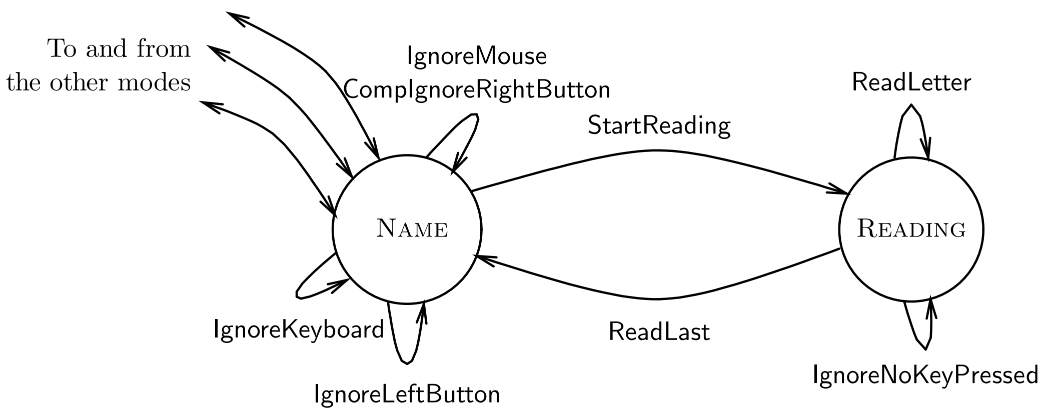


Figure 5.4: X-machine-let showing the replacement for Name

So, any solution is probably going to have to involve a richer data set, that can store the relevant details of a user's input. On the other hand, it would be pity to have to redefine the whole of the machine—most of the transition function definitions are fine, and will not use the enrichments of the data set even when they are available. Also, for simple but common procedures, such as reading in text sequences from the keyboard, it would be very useful to re-use a standard model, such as the one presented in example 4.2.2 of chapter 4.

5.6 *X*-machine model of the tool revisited.

By using the idea of stream-X-machine-lets from section 4.4, $\mathcal{M}_1$ can be refined into an improved version, $\mathcal{M}_2$. The new parts are fully-deterministic stream-X-machine-lets that are complete with respect to Y, and are added in by general refinement.

5.6.1 Name, revisited.

In order to properly model the user inputting a new name, the state Name needs to be replaced with an X-machine-let, similar to the one from example 4.2.2 of chapter 4.

All that is needed is a single new state, Reading, and some new function definitions. Figure 5.4 shows how they fit together. The new name can be up to n characters in length.

The function to change to Reading mode, StartReading:

$$\mathsf{StartReading} = \lambda(G, ((P,T,i,o),\mu,\mathcal{L}),(c,m,k) :: S) : X \bullet$$
$$\begin{cases} ((c,\mathcal{L}',i,o,\mu) :: G, \\ \quad ((P,t,i,o),\mu,\mathcal{L}',name),S) & \text{if } \mathit{LeftPressed}(m) \wedge \mathit{Overlap}(\{c\},\mathcal{L}) \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$name = \begin{cases} \mathcal{L}.\pi(\mathit{PMiddle}(c)) & \text{if } \mathit{PlcOverlap}(\{c\},\mathcal{L}) \\ \mathcal{L}.\tau(\mathit{TMiddle}(c)) & \text{if } \mathit{TransOverlap}(\{c\},\mathcal{L}) \end{cases}$$

$$\mathcal{L}' \quad = (\mathcal{L}.\pi, \mathcal{L}.\tau, \mathcal{L}.\nu, \langle\rangle)$$

The fundamental data set has been expanded for the purposes of the X-machine-let, so that

$$X_{new} = X_{old} \uplus (\Gamma^* \times (\textsc{Net} \times \textsc{Marking} \times \textsc{Locations} \times \textsc{Names}) \times \Sigma^*).$$

This allows the (original) name of the place or transition being moved to be stored while the characters making up the new name are read in.

The function to read in a single letter, ReadLetter:

$$\mathsf{ReadLetter} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}, \mathit{name}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}', \mathit{name}), S) & \text{if } k \in \{a, \ldots, z\} \wedge \\ & \text{len}(\mathcal{L}.\delta) < n-1 \\ & \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \mathcal{L}.\delta \frown \langle k \rangle)$$

In other words, ReadLetter reads in the next character, and adds it to those already entered.

The function to deal with *Return* or the *n*th character, ReadLast:

$$\mathsf{ReadLast} = \lambda(G, ((P, t, i, o), \mu, \mathcal{L}, \mathit{oldname}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', i', o', \mu') :: G, & \\ \quad ((P', T', i', o'), \mu', \mathcal{L}'), S) & \text{if } k = \mathit{Return} \vee \\ & (k \in \{a, \ldots, z\} \wedge \text{len}(\mathcal{L}.\delta) = n-1) \\ & \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathit{newname} = \begin{cases} \mathcal{L}.\delta & \text{if } k = \mathit{Return} \\ \mathcal{L}.\delta \frown \langle k \rangle & \text{if } k \in \{a, \ldots, z\} \end{cases}$$

$$P' = \begin{cases} (P \setminus \{\mathit{oldname}\}) \cup \{\mathit{newname}\} & \text{if } \mathit{oldname} \in P \\ P & \text{if } \mathit{oldname} \in T \end{cases}$$

$$T' = \begin{cases} T & \text{if } \mathit{oldname} \in P \\ (T \setminus \{\mathit{oldname}\}) \cup \{\mathit{newname}\} & \text{if } \mathit{oldname} \in T \end{cases}$$

$$\mu' = \begin{cases} (\{\mathit{oldname}\} \lhd \mu) \oplus & \text{if } \mathit{oldname} \in P \\ \quad \{\mathit{newname} \mapsto \mu(\mathit{oldname})\} & \\ \mu & \text{if } \mathit{oldname} \in T \end{cases}$$

$$i' = \begin{cases} \lambda\, t : \textsc{Transitions} \bullet (\{\mathit{oldname}\} \lhd i(t)) \oplus & \text{if } \mathit{oldname} \in P \\ \quad \{\mathit{newname} \mapsto (i(t))(\mathit{oldname})\} & \\ (\{\mathit{oldname}\} \lhd i) \oplus \{\mathit{newname} \mapsto i(\mathit{oldname})\} & \text{if } \mathit{oldname} \in T \end{cases}$$

$$o' = \begin{cases} \lambda\, t : \textsc{Transitions} \bullet (\{\mathit{oldname}\} \lhd o(t)) \oplus & \text{if } \mathit{oldname} \in P \\ \quad \{\mathit{newname} \mapsto (o(t))(\mathit{oldname})\} & \\ (\{\mathit{oldname}\} \lhd o) \oplus \{\mathit{newname} \mapsto o(\mathit{oldname})\} & \text{if } \mathit{oldname} \in T \end{cases}$$

$$\begin{array}{lll}
\mathcal{L}' & = \begin{cases}(pl, \mathcal{L}.\tau, \mathcal{L}.\nu, \mathcal{L}.\delta) & \text{if } \mathit{oldname} \in P \\ (\mathcal{L}.\pi, tr, \mathcal{L}.\nu, \mathcal{L}.\delta) & \text{if } \mathit{oldname} \in T\end{cases} \\
pl & = \lambda\, c : \textsc{Positions} \bullet \begin{cases}\mathcal{L}.\pi(c) & \text{if } \mathcal{L}.\pi(c) \neq \mathit{oldname} \\ \mathit{newname} & \text{if } \mathcal{L}.\pi(c) = \mathit{oldname}\end{cases} \\
tr & = \lambda\, c : \textsc{Positions} \bullet \begin{cases}\mathcal{L}.\tau(c) & \text{if } \mathcal{L}.\tau(c) \neq \mathit{oldname} \\ \mathit{newname} & \text{if } \mathcal{L}.\tau(c) = \mathit{oldname}\end{cases}
\end{array}$$

The net effect of all this is to replace all occurrences of *oldname* in the net with *newname*.

The function to ignore irrelevant keyboard input, IgnoreNoKeyPressed**:**

$$\mathsf{IgnoreNoKeyPressed} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}, \mathit{name}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases}((c, \mathcal{L}, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu) :: G, & \text{if } \mathit{NoKeyPressed}(k) \\ \quad (\mathcal{P}, \mu, \mathcal{L}, \mathit{name}), S) & \\ \perp & \text{otherwise}\end{cases}$$

5.6.2 Arc, revisited.

A similar approach can be used to deal properly with the addition and removal of arcs. The main difference is that there are two X-machine-lets, one for adding, one for removing, instead of just one.

Firstly, I will describe the X-machine-let for adding a new arc:

The function to start adding an arc, StartArcAdd**:**

$$\mathsf{StartArcAdd} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases}((c, \mathcal{L}, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu) :: G, & \text{if } \mathit{LeftPressed}(m) \wedge \mathit{Overlap}(\{c\}, \mathcal{L}) \\ \quad (\mathcal{P}, \mu, \mathcal{L}, \mathit{startname}), S) & \\ & \\ \perp & \text{otherwise}\end{cases}$$

where

$$\mathit{startname} = \begin{cases}\mathcal{L}.\pi(\mathit{PMiddle}(c)) & \text{if } \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ \mathcal{L}.\tau(\mathit{TMiddle}(c)) & \text{if } \mathit{TransOverlap}(\{c\}, \mathcal{L})\end{cases}$$

As in the case of the Name X-machine-let, the fundamental data set is expanded to allow the name of the place or transition at the beginning of the new arc to be stored whilst the transition or place at the end is read.

The function to finish adding an arc, EndArcAdd**:**

$$\mathsf{EndArcAdd} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}, \mathit{startname}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases}((c, \mathcal{L}, i', o', \mu) :: G, & \\ \quad ((P, T, i', o'), \mu, \mathcal{L}), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & ((\mathit{startname} \in P) \wedge (\mathit{endname} \in T) \vee \\ & (\mathit{startname} \in T) \wedge (\mathit{endname} \in P)) \\ & \textit{i.e}, \text{ if the left mouse button is pressed in a} \\ & \text{suitable place or transition for the arc to be} \\ & \text{finished off} \\ \perp & \text{otherwise}\end{cases}$$

where

$$endname = \begin{cases} \mathcal{L}.\pi(PMiddle(c)) & \text{if } PlcOverlap(\{c\}, \mathcal{L}) \\ \mathcal{L}.\tau(TMiddle(c)) & \text{if } TransOverlap(\{c\}, \mathcal{L}) \end{cases}$$

$$i' = \begin{cases} i & \text{if } endname \in P \\ i \oplus \{endname \mapsto (i(endname) \oplus \\ \quad \{startname \mapsto (i(endname))(startname) + 1\})\} & \text{if } endname \in T \end{cases}$$

$$o' = \begin{cases} o \oplus \{startname \mapsto (o(startname) \oplus & \text{if } endname \in P \\ \quad \{endname \mapsto (o(startname))(endname) + 1\})\} & \\ o & \text{if } endname \in T \end{cases}$$

The function to ignore irrelevant mouse and keyboard input, $\mathsf{ArcAddIgnore}$**:**

$$\mathsf{ArcAddIgnore} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}, name), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}, name), S) & \text{if } RightPressed(m) \vee NonePressed(m) \vee \\ & (LeftPressed(m) \wedge \\ & ((startname, endname \in P) \vee \\ & (startname, endname \in T))) \\ \bot & \text{otherwise} \end{cases}$$

where

$$endname = \begin{cases} \mathcal{L}.\pi(PMiddle(c)) & \text{if } PlcOverlap(\{c\}, \mathcal{L}) \\ \mathcal{L}.\tau(TMiddle(c)) & \text{if } TransOverlap(\{c\}, \mathcal{L}) \end{cases}$$

Secondly, here are the functions for the X-machine-let for removing arcs.

The function to start removing an arc, $\mathsf{StartArcDel}$**:**

$$\mathsf{StartArcDel} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu) :: G, & \text{if } RightPressed(m) \wedge Overlap(\{c\}, \mathcal{L}) \\ \quad (\mathcal{P}, \mu, \mathcal{L}, startname), S) & \\ \bot & \text{otherwise} \end{cases}$$

where

$$startname = \begin{cases} \mathcal{L}.\pi(PMiddle(c)) & \text{if } PlcOverlap(\{c\}, \mathcal{L}) \\ \mathcal{L}.\tau(TMiddle(c)) & \text{if } TransOverlap(\{c\}, \mathcal{L}) \end{cases}$$

This uses the same expanded data set as the functions that add a new arc.

The function to finish removing an arc, $\mathsf{EndArcDel}$**:**

$$\mathsf{EndArcDel} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}, startname), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, i', o', \mu) :: G, & \\ \quad ((P, T, i', o'), \mu, \mathcal{L}), S) & \text{if } RightPressed(m) \wedge \\ & ((startname \in P) \wedge (endname \in T) \vee \\ & (startname \in T) \wedge (endname \in P)) \\ & i.e, \text{ if the right mouse button is pressed in a} \\ & \text{suitable place or transition for the arc to be} \\ & \text{finished off} \\ \bot & \text{otherwise} \end{cases}$$

where

$$endname = \begin{cases} \mathcal{L}.\pi(PMiddle(c)) & \text{if } PlcOverlap(\{c\}, \mathcal{L}) \\ \mathcal{L}.\tau(TMiddle(c)) & \text{if } TransOverlap(\{c\}, \mathcal{L}) \end{cases}$$

$$\begin{aligned} i' &= \begin{cases} i & \text{if } endname \in P \\ i \oplus \{endname \mapsto (i(endname) \oplus \\ \quad \{startname \mapsto ni\})\} & \text{if } endname \in T \end{cases} \\ ni &= \begin{cases} 0 & \text{if } (i(endname))(startname) \leq 1 \\ (i(endname))(startname) - 1 & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} o' &= \begin{cases} o \oplus \{startname \mapsto (o(startname) \oplus & \text{if } endname \in P \\ \quad \{endname \mapsto no\})\} & \\ o & \text{if } endname \in T \end{cases} \\ no &= \begin{cases} 0 & \text{if } (o(startname))(endname) \leq 1 \\ (o(startname))(endname) - 1 & \text{otherwise} \end{cases} \end{aligned}$$

The function to ignore irrelevant mouse and keyboard input, $\mathsf{ArcDelIgnore}$**:**

$$\mathsf{ArcDelIgnore} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}, name), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}, name), S) & \text{if } LeftPressed(m) \vee NonePressed(m) \vee \\ & (RightPressed(m) \wedge \\ & ((startname, endname \in P) \vee \\ & (startname, endname \in T))) \\ \bot & \text{otherwise} \end{cases}$$

where

$$endname = \begin{cases} \mathcal{L}.\pi(PMiddle(c)) & \text{if } PlcOverlap(\{c\}, \mathcal{L}) \\ \mathcal{L}.\tau(TMiddle(c)) & \text{if } TransOverlap(\{c\}, \mathcal{L}) \end{cases}$$

The ways these functions are connected into the original X-machine together with two new states, ArcAdd, and ArcDel, are shown in Figure 5.5.

5.6.3 Move, revisited.

The function to start moving a place or transition, $\mathsf{StartMove}$**:**

$$\mathsf{StartMove} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu) :: G, & \text{if } LeftPressed(m) \wedge Overlap(\{c\}, \mathcal{L}) \\ \quad (\mathcal{P}, \mu, \mathcal{L}, middle), S) & \\ \bot & \text{otherwise} \end{cases}$$

where

$$middle = \begin{cases} PMiddle(c) & \text{if } PlcOverlap(\{c\}, \mathcal{L}) \\ TMiddle(c) & \text{if } TransOverlap(\{c\}, \mathcal{L}) \end{cases}$$

This X-machine-let uses a different extension to the fundamental data set:

$$X_{new} = X_{old} \uplus (\Gamma^* \times (\textsc{Net} \times \textsc{Marking} \times \textsc{Locations} \times \textsc{Positions}) \times \Sigma^*).$$

In this case it is only necessary to store the original middle position of the place or transition being moved.

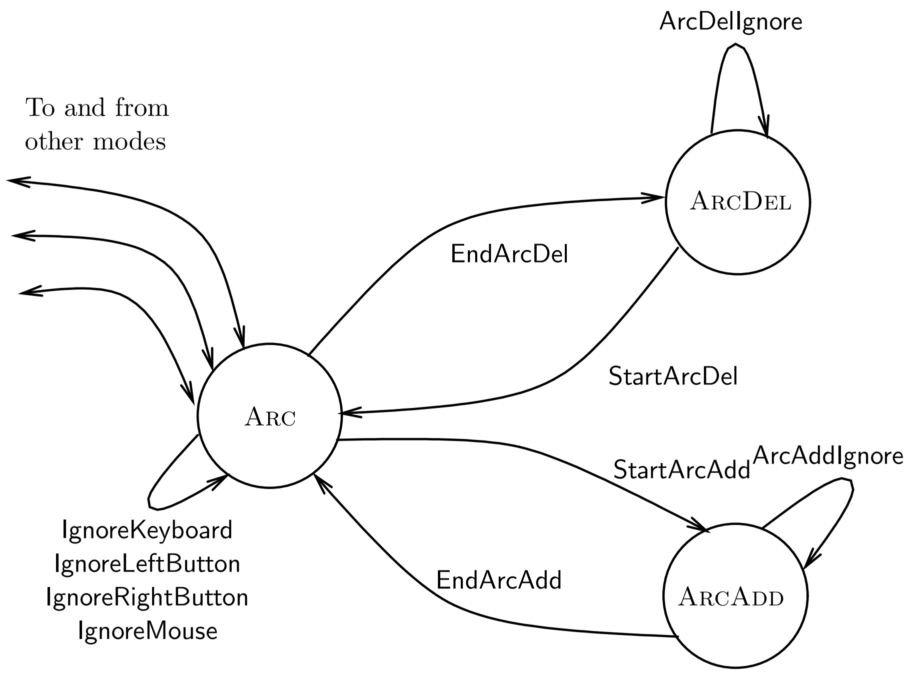


Figure 5.5: X-machine-let showing the replacements for Arc

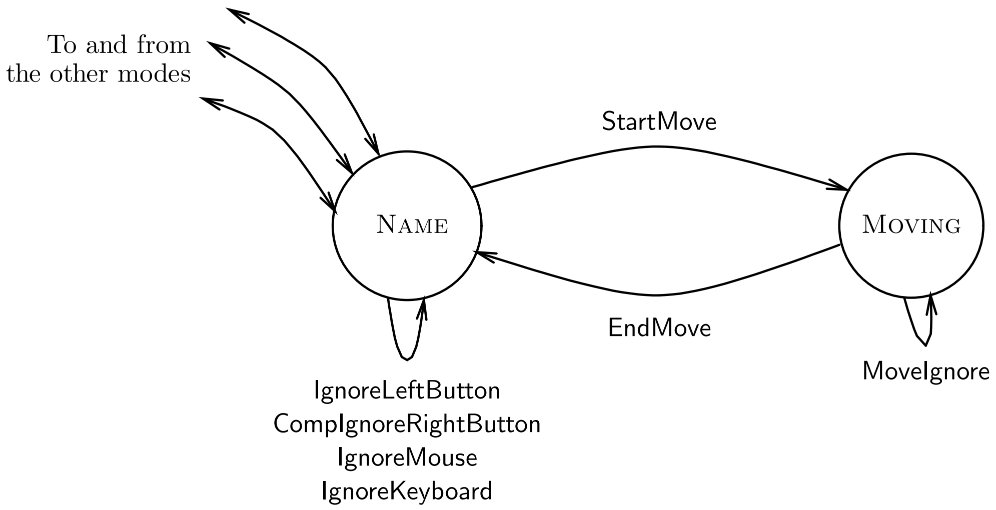


Figure 5.6: X-machine-let to move places and transitions.

The function to finish moving a place or transition, EndMove:

$$\mathsf{EndMove} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}, \mathit{middle}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu, \mathcal{L}') :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & ((\mathit{middle} \in \mathrm{dom}(\mathcal{L}.\pi) \wedge \\ & \mathit{PlcOverlap}(\{c\}, \mathcal{L}^\dagger)) \vee \\ & (\mathit{middle} \in \mathrm{dom}(\mathcal{L}.\tau) \wedge \\ & \mathit{TransOverlap}(\{c\}, \mathcal{L}^\dagger))) \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathcal{L}^\dagger &= \begin{cases} (pl, \mathcal{L}.\tau, \mathcal{L}.\nu, \mathcal{L}.\delta) & \text{if } \mathit{middle} \in \mathrm{dom}(\mathcal{L}.\pi) \\ (\mathcal{L}.\pi, tr, \mathcal{L}.\nu, \mathcal{L}.\delta) & \text{if } \mathit{middle} \in \mathrm{dom}(\mathcal{L}.\tau) \end{cases} \\ pl &= \{\mathit{middle}\} \lhd \mathcal{L}.\pi \\ tr &= \{\mathit{middle}\} \lhd \mathcal{L}.\tau \end{aligned}$$

$$\begin{aligned} \mathcal{L}' &= \begin{cases} (pl', \mathcal{L}.\tau, \mathcal{L}.\nu, \mathcal{L}.\delta) & \text{if } \mathit{middle} \in \mathrm{dom}(\mathcal{L}.\pi) \\ (\mathcal{L}.\pi, tr', \mathcal{L}.\nu, \mathcal{L}.\delta) & \text{if } \mathit{middle} \in \mathrm{dom}(\mathcal{L}.\tau) \end{cases} \\ pl' &= pl \oplus \{c \mapsto \mathcal{L}.\pi\} \\ tr' &= tr \oplus \{c \mapsto \mathcal{L}.\tau\} \end{aligned}$$

The definition of $\mathcal{L}^\dagger$ is necessary in order to check whether the new position for the object being moved is too close to any of the other places or transitions. The definition is complicated by the need to take into account the fact that the space previously occupied by the object is now available again.

The function to ignore irrelevant input during a move, MoveIgnore:

$$\mathsf{MoveIgnore} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}, \mathit{middle}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{P}.\textsc{InArcs}, \mathcal{P}.\textsc{OutArcs}, \mu, \mathcal{L}) :: G, & \\ \quad (\mathcal{P}, \mu, \mathcal{L}, \mathit{middle}), S) & \text{if } \mathit{RightPressed}(m) \vee \mathit{NonePressed}(m) \vee \\ & (\mathit{LeftPressed}(m) \wedge \neg\, \mathit{Overlap}(\{c\}, \mathcal{L}^\dagger)) \\ \perp & \text{otherwise} \end{cases}$$

where $\mathcal{L}^\dagger$ is as defined above.

Figure 5.6 shows how the functions connect up for the moving X-machine-let.

Chapter 6

Tests from X-machines

6.1 Faults in X-machines.

Given a specification S modelled by $\mathcal{S}$, a fully-deterministic stream-X-machine that is complete w.r.t. Y, what are the faults that can occur in implementing it as I, modelled by $\mathcal{I}$? (Without loss of generality, $\mathcal{I}$ can be assumed to be exactly the same as $\mathcal{S}$ apart from faults that it might contain.)

$$\begin{aligned}\mathcal{S} &= (X, Y, Z, \alpha_{\mathcal{S}}, \beta_{\mathcal{S}}, Q_{\mathcal{S}}, \Phi, F_{\mathcal{S}}, I_{\mathcal{S}}, T_{\mathcal{S}})\\ \mathcal{I} &= (X, Y, Z, \alpha_{\mathcal{I}}, \beta_{\mathcal{I}}, Q_{\mathcal{I}}, \Phi, F_{\mathcal{I}}, I_{\mathcal{I}}, T_{\mathcal{I}})\end{aligned}$$

$$X = \Gamma^* \times M \times \Sigma^*$$

1. There could be missing states in $\mathcal{I}$ compared to $\mathcal{S}$ (which would lead to misdirected transitions), *i.e.*

   $$Q_{\mathcal{I}} \subset Q_{\mathcal{S}}.$$

2. There could by missing or mis-directed transitions in $\mathcal{I}$, *i.e.*

   $$\exists\, q \in (Q_{\mathcal{S}} \cap Q_{\mathcal{I}}) \bullet [\mathrm{dom}(F_{\mathcal{S}}(q)) \supseteq \mathrm{dom}(F_{\mathcal{I}}(q))] \vee [F_{\mathcal{S}}(q) \neq F_{\mathcal{I}}(q)].$$

3. There could be transitions that have faulty functions in $\mathcal{I}$, *i.e.*

   $$\begin{aligned}&\exists\, \phi_{\mathcal{S}}, \phi_{\mathcal{I}} \in \Phi \bullet \phi_{\mathcal{S}} \neq \phi_{\mathcal{I}} \wedge \mathrm{dom}(\phi_{\mathcal{S}}) = \mathrm{dom}(\phi_{\mathcal{I}}) \wedge\\ &\quad (\exists\, q \in (Q_{\mathcal{S}} \cap Q_{\mathcal{I}}) \bullet F_{\mathcal{S}}(q, \phi_{\mathcal{S}}) = F_{\mathcal{I}}(q, \phi_{\mathcal{I}}))\end{aligned}$$

4. There could be extra transitions in $\mathcal{I}$, *i.e.*

   $$\exists\, q \in (Q_{\mathcal{S}} \cap Q_{\mathcal{I}}) \bullet [\mathrm{dom}(F_{\mathcal{S}}(q)) \subset \mathrm{dom}(F_{\mathcal{I}}(q))].$$

5. There could be extra states in $\mathcal{I}$, *i.e.*

   $$Q_{\mathcal{I}} \supset Q_{\mathcal{S}}.$$

Of these, the “extra states” faults, the “extra transitions” faults and the “faulty transition function” faults are the most problematic.

However, it is worth bearing in mind that extra transitions cannot exist in isolation. Adding an extra transition to $\mathcal{I}$ (compared to $\mathcal{S}$) requires that other transitions be missing, mis-directed or faulty, or $\mathcal{I}$ will not be fully-deterministic and complete w.r.t. Y.

6.1.1 Missing states.

By using a state cover set for $\mathcal{S}$ on $\mathcal{I}$, every state q in $\mathcal{S}$ that is also in $\mathcal{I}$ will be visited, and any missing states will be revealed.

6.1.2 Missing or mis-directed transitions.

By using a transition cover set for $\mathcal{S}$ on $\mathcal{I}$, every transition ϕ in $\mathcal{S}$ that is also in $\mathcal{I}$ will be used, and any missing or mis-directed transitions will be revealed. As with missing states, it is necessary to use the characterisation set to check that each transition end in the expected state.

A transition cover set must include a state cover set.

6.2 Transitions with faulty functions.

These are harder to deal with. However, since $\mathcal{S}$ is a fully-deterministic stream-X-machine that is complete w.r.t. Y, every $\phi \in \Phi$ is of the form:

$$\phi = (g, m, h :: s) : \Gamma^* \times M \times \Sigma^* \bullet \begin{cases} (\gamma_1(m,h) :: g, \phi_1(m,h), s) & \text{if } c_1(m,h) \\ (\gamma_2(m,h) :: g, \phi_2(m,h), s) & \text{if } c_2(m,h) \\ \quad\vdots & \\ (\gamma_n(m,h) :: g, \phi_n(m,h), s) & \text{if } c_n(m,h) \\ \quad\perp & \text{if } c_\perp(m,h) \\ & \quad (i.e. \text{ otherwise}) \end{cases}$$

where the ϕ_is and γ_is are functions of m and h ($i.e.$ $\phi_i : M \times \Sigma \to M$ and $\gamma_i : M \times \Sigma \to \Gamma$). The c_i ($i \in \{1..n\}$) are mutually exclusive conditions on m and h ($i.e.$ $c_i : M \times \Sigma \to \mathbb{B}$), and so, for each state $q \in Q$, partition $\mathrm{dom}(\phi) \cap \mathrm{XAttainable}(q)$. Since the domains of the $\phi \in \mathrm{dom}(F(q))$ partition $\mathrm{XAttainable}(q)$ (see Lemma 4.2.9), the c_i of each ϕ produce a more detailed partition of $\mathrm{XAttainable}(q)$.

The last condition is the negation of all the others: $c_\perp(m,h) \Leftrightarrow \neg\ (c_1(m,h) \vee c_2(m,h) \vee \ldots \vee c_n(m,h))$.

Notation 6.2.1 For ϕ of the form shown above, and $i \in \{1, 2, \ldots, n, \perp\}$, call ϕ's equivalence classes C_i^ϕ after the conditions c_i that produce them. So

$$\left[(G, m, S) \in C_i^\phi\right] \Leftrightarrow [c_i(m, \mathit{head}\, S) = \text{true}] .$$

◇

Definition 6.2.2 For $\phi \in \Phi$, the equivalence classes are generated by:

$$\mathsf{Classes}(\phi) = \{C_1^\phi, C_2^\phi, \ldots, C_n^\phi\}.$$

Notice that $C_\perp^\phi$ is not included. ◇

Definition 6.2.3 Given $\mathcal{M}$, a fully-deterministic stream-X-machine that is complete with respect to Y, a set of input sequences $T_0 \subseteq \Sigma^*$ is a *basic partition-based transition cover set* (BPBTC) if it contains two test sequences $t_{C_i^\phi}$ and $t'_{C_i^\phi}$ for each C_i^ϕ of each function $\phi \in \Phi$ each time it is used in $\mathcal{M}$. *i.e.*

$$\begin{aligned} &\forall\, q \in Q^{\mathcal{M}} \bullet \forall\, \phi \in \mathrm{dom}(F(q)) \bullet \forall\, C \in \mathsf{Classes}(\phi) \bullet \exists\, t \frown \langle s \rangle \in T_0 \bullet \\ &\quad \left[q_0 \xrightarrow{q_0|_\Phi(m_0,t)} q\right] \wedge \left[\phi = q_0 \parallel_\Phi (m_0, t \frown \langle s \rangle)\right] \wedge \\ &\quad\quad [(q_0 |_\Gamma (m_0, t), q_0 \parallel_M (m_0, t), s) \in C] . \end{aligned}$$

where $t_{C_i^\phi} = t \frown \langle s \rangle$ and $t'_{C_i^\phi} = t$. ◇

A BPBTC set includes a transition cover set.

A partition cover will reveal all the faults for which the domains of each ϕ_i sub-function of each $\phi \in \Phi$ are revealing. However, even if the ϕ_i are very simple, it is unlikely that a single test case for each one will reveal all the possible faults.

Example 6.2.4 This example shows how to build a basic partition-based transition cover set for the machine given in example 4.2.2 (page 46). Assume that the maximum string length, n, is 5. There are several steps to go through.

Partition the functions into their equivalence classes:

Recall that $\Phi = \{\mathsf{r}, \mathsf{r}', \mathsf{end}, \mathsf{Return}\}$.

1. $\mathsf{Classes}(\mathsf{r})$ has the following parts:
   - $C_1^{\mathsf{r}} = \{(G, m, S) : X \mid \mathit{head}\, S \in \{a, \ldots, z\} \wedge \mathrm{len}(m) < 4\}$,
2. $\mathsf{Classes}(\mathsf{r}')$ has the following parts:
   - $C_1^{\mathsf{r}'} = \{(G, m, S) : X \mid \mathit{head}\, S \in \{a, \ldots, z\} \wedge \mathrm{len}(m) > 4\}$,
3. $\mathsf{Classes}(\mathsf{end})$ has the following parts:
   - $C_1^{\mathsf{end}} = \{(G, m, S) : X \mid \mathit{head}\, S \in \{a, \ldots, z\} \wedge \mathrm{len}(m) = 4\}$,
4. $\mathsf{Classes}(\mathsf{Return})$ has the following parts:
   - $C_1^{\mathsf{Return}} = \{(G, m, S) : X \mid h = \mathit{Return}\}$,

Generate the test cases that cover these partitions. Inputs are given in two parts: $i_1 \frown \langle i_2 \rangle$. The first part, i_1 corresponds to a sequence that prepares the machine for the second part, i_2, which is an element corresponding to the appropriate equivalence class.

Similarly, outputs are given in two parts: $o_1 \frown \langle o_2 \rangle$. The first part, o_1, is the output due to the first part of the input, i_1, and the second part, o_2 is the single output due to the second part of the input, i_2.

Thus the two test cases can be found: $t = i_1 \frown \langle i_2 \rangle$, leading to output $o_1 \frown \langle o_2 \rangle$, and $t' = i_1$, leading to output o_1.

$Q = \{\textsc{Read}, \textsc{End}\}$.

1. $F(\textsc{Read}) = \{\mathsf{r} \mapsto \textsc{Read}, \mathsf{end} \mapsto \textsc{End}, \mathsf{Return} \mapsto \textsc{End}\}$
   (a) $\textsc{Read} \xrightarrow{\mathsf{r}} \textsc{Read}$. There has to be a test case for each equivalence class of r.
   - C_1^{r} leads to the following test case:
     $$t_{C_1^{\mathsf{r}}} = \langle\rangle \frown \langle a \rangle,$$
     which has expected output of $\langle\rangle \frown \langle a \rangle$.

   (b) $\textsc{Read} \xrightarrow{\mathsf{end}} \textsc{End}$.
   - C_1^{end} leads to the following test case:
     $$t_{C_1^{\mathsf{end}}} = \langle a.a.a.a \rangle \frown \langle a \rangle,$$
     which is expected to result in $\langle a.a.a.a \rangle \frown \langle a \rangle$ as output.
     output.

(c) $\text{Read} \xrightarrow{\mathsf{Return}} \text{End}$.

- C_1^{Return} leads to the following test case:

  $$t_{C_1^{\mathsf{Return}}} = \langle\rangle \frown \langle \mathit{Return} \rangle,$$

  which is expected to result in $\langle\rangle \frown \langle\rangle$ as output.

2. $F(\text{End}) = \{\mathsf{Return} \mapsto \text{End}, \mathsf{r}' \mapsto \text{End}\}$.

(a) $\text{End} \xrightarrow{\mathsf{Return}} \text{End}$.

- C_1^{Return} leads to the following test case:

  $$t_{C_1^{\mathsf{Return}}} = \langle a.a.a.a.a \rangle \frown \langle \mathit{Return} \rangle,$$

  which is expected to result in $\langle a.a.a.a.a \rangle \frown \langle\rangle$ as output.
  output.

(b) $\text{End} \xrightarrow{\mathsf{r}'} \text{End}$.

- $C_1^{\mathsf{r}'}$ leads to the following test case:

  $$t_{C_1^{\mathsf{r}'}} = \langle a.a.a.a.a \rangle \frown \langle a \rangle,$$

  which is expected to lead to $\langle a.a.a.a.a \rangle \frown \langle a \rangle$ as output.
  output.

◇

Detailed partition covers.

The example above reveals limitations in the use of basic partition-based transition covers. The equivalence classes are very "coarse", in that, even for a simple example, they are not revealing for many faults.

Definition 6.2.5 A more detailed partition can be constructed by considering the conditions, c_i, of the functions in more detail. If they are compound predicates in disjunctive normal form, they can be written as follows:

$$c_i(m,h) = b_{i1}(m,h) \vee b_{i2}(m,h) \vee \ldots \vee b_{il}(m,h)$$

where each b_{ij} is a conjunction dependent on (m,h).

The b_{ij} lead to up to 2^l different equivalence classes on X corresponding to predicates of the form

$$\left[\bigwedge_{j \in J} b_{ij}\right] \wedge \left[\bigwedge_{j \in J'} \neg\, b_{ij}\right]$$

where $J \cup J' = \{1, \ldots l\}$ and $J \cap J' = \varnothing$. Some of the potential classes might be empty, which is why there are "up to" 2^l classes, not "exactly" 2^l classes.

Clearly, test cases can be defined for these classes in the same way as for the classes of definition 6.2.2. Call these the *disjoint-classes* of ϕ, $\mathsf{DClasses}(\phi)$. ◇

Definition 6.2.6 The concept of disjoint-classes leads naturally to a more detailed coverage, defined in the same way as a basic partition-based transition cover (definition 6.2.3), but called a *disjoint partition-based transition cover.*

Given $\mathcal{M}$, a fully-deterministic stream-X-machine that is complete w.r.t. Y, a set of input sequences $T_0 \subseteq \Sigma^*$ is a *disjoint partition-based transition cover set* (disjoint PBTC) if it contains two test sequences t and t' for each disjoint partition, $\mathcal{D}$, of each function $\phi \in \Phi$ each time it is used in $\mathcal{M}$. *i.e.*

$$\begin{aligned}
&\forall\, q \in Q^{\mathcal{M}} \bullet \forall\, \phi \in \mathrm{dom}(F(q)) \bullet \forall\, D \in \mathsf{DClasses}(\phi) \bullet \exists\, t \frown \langle s \rangle \in T_0 \bullet \\
&\quad \left[q_0 \xrightarrow{q_0 |_\Phi (m_0, t)} q \right] \wedge \left[\phi = q_0 \parallel_\Phi (m_0, t \frown \langle s \rangle) \right] \wedge \\
&\qquad [(q_0 |_\Gamma (m_0, t), q_0 \parallel_M (m_0, t), s) \in D]\,.
\end{aligned}$$

General partition covers.

Basic and disjoint partition-based transition covers are generated in the same way once their respective partitions of X are defined. The principle can be extended to *any* partition of X.

Definition 6.2.7 Let $\mathcal{M}$ be a stream-X-machine with the usual components. A *test-partition generator* for $\mathcal{M}$ is a function

$$\mathcal{G} : \Phi \to \mathbb{P}[\mathbb{P}(X)]$$

that partitions the set X into subsets suitable for revealing faults in each of the members of Φ. $\diamond$

Definition 6.2.8 A partition $\mathcal{P} \in \mathcal{G}(\phi)$ *recognises* its members; *i.e.*

$$\mathcal{P} \textit{ recognizes } x \Leftrightarrow x \in \mathcal{P}.$$

$\diamond$

Definition 6.2.9 Given a test-partition generator for $\mathcal{M}$, $\mathcal{G}$, and a transition $q_1 \xrightarrow{\phi} q_2$ in $\mathcal{M}$, then a $\mathcal{G}$ *partition-based test* for the transition is a set of pairs as follows:

Each partition, $\mathcal{P} \in \mathcal{G}(\phi)$ either leads to a pair of test cases or to no test cases, according to whether or not a suitable preparation sequence exists. *i.e*, whether

$$\mathrm{XAttainable}(q_1) \cap \mathcal{P} = \varnothing.$$

If the intersection is empty, then no member of $\mathcal{P}$ is attainable in q_1, and $\mathcal{P}$ is said to *unreachable* for this particular transition. In which case there is no pair of test cases for this partition.

If the intersection is not empty, then some members of $\mathcal{P}$ are attainable in q_1, and $\mathcal{P}$ is *reachable* for this transition. In which case

$$\begin{aligned}
&\exists\, t \in \Sigma^*;\ \exists\, h \in \Sigma \bullet \\
&\quad [q_0 \parallel_Q (m_0, t) = q_1] \wedge [q_0 \parallel_M (m_0, t) = m] \wedge [q_0 |_\Gamma (m_0, t) = G] \wedge \\
&\qquad [(G, m, h) \in \mathcal{P}] \wedge [q_0 \in I]
\end{aligned}$$

The *preparation* test case is one such t, and the *transition* test case is $t \frown \langle h \rangle$, using one such h. $\diamond$

Definition 6.2.10 Let $\mathcal{M}$ be a stream-X-machine in the usual notation.

Given $\mathcal{G}$, a test-partition generator, then the $\mathcal{G}$ *partition-based transition cover* (or $\mathcal{G}$-PBTC) is a test set consisting of the union of the $\mathcal{G}$ partition based tests for each transition in $\mathcal{M}$. $\diamond$

Example 6.2.11 Returning, once again to $\mathcal{M}_{4.2.2}$ of example 4.2.2 that was further examined in example 6.2.4, reconsider it in the light of the new definitions. This time, let $n = 6$.

The test-partition generator, $\mathcal{G}$: This is based on the equivalence classes of example 6.2.4:

1. $\mathcal{G}(\mathsf{r})$ has the following partitions:

$$\begin{aligned}
\mathcal{P}_1 &= \{(G, m, S) \in X \mid \text{len}(m) = 4 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},\\
\mathcal{P}_2 &= \{(G, m, S) \in X \mid 1 < \text{len}(m) < 4 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},\\
\mathcal{P}_3 &= \{(G, m, S) \in X \mid \text{len}(m) = 1 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},\\
\mathcal{P}_4 &= \{(G, m, S) \in X \mid \text{len}(m) = 0 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},
\end{aligned}$$

2. $\mathcal{G}(\mathsf{r}')$ has the following partitions:

$$\begin{aligned}
\mathcal{P}_1 &= \{(G, m, S) \in X \mid \text{len}(m) = 6 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},\\
\mathcal{P}_2 &= \{(G, m, S) \in X \mid \text{len}(m) = 7 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},\\
\mathcal{P}_3 &= \{(G, m, S) \in X \mid \text{len}(m) > 7 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},
\end{aligned}$$

3. $\mathcal{G}(\mathsf{end})$ has the following partitions:

$$\mathcal{P}_1 = \{(G, m, S) \in X \mid \text{len}(m) = 5 \wedge \mathit{head}\, S \in \{a, \ldots, z\}\},$$

4. $\mathcal{G}(\mathsf{Return})$ has the following partitions:

$$\begin{aligned}
\mathcal{P}_1 &= \{(G, m, S) \in X \mid \text{len}(m) = 0 \wedge \mathit{head}\, S \in \{\mathit{Return}\}\},\\
\mathcal{P}_2 &= \{(G, m, S) \in X \mid 0 < \text{len}(m) < 5 \wedge \mathit{head}\, S \in \{\mathit{Return}\}\},\\
\mathcal{P}_3 &= \{(G, m, S) \in X \mid \text{len}(m) = 5 \wedge \mathit{head}\, S \in \{\mathit{Return}\}\},\\
\mathcal{P}_4 &= \{(G, m, S) \in X \mid \text{len}(m) = 6 \wedge \mathit{head}\, S \in \{\mathit{Return}\}\},\\
\mathcal{P}_5 &= \{(G, m, S) \in X \mid \text{len}(m) = 7 \wedge \mathit{head}\, S \in \{\mathit{Return}\}\},\\
\mathcal{P}_6 &= \{(G, m, S) \in X \mid \text{len}(m) > 7 \wedge \mathit{head}\, S \in \{\mathit{Return}\}\},
\end{aligned}$$

Generate the test cases that cover these partitions. Actually, only a few partitions will be considered, as the generation of test cases is almost entirely mechanical once the partitions have been defined.

Test cases are given in two parts, $t : \Sigma^*$, the preparation test case and $h : \Sigma$, so that $t \frown \langle h \rangle$ is the transition test case. Similarly, the output is given in the form $o : \Gamma^*$ and $g : \Gamma$, where $q_0 \mid_\Gamma (t) = o$ and $q_0 \mid_\Gamma (t \frown \langle h \rangle) = g :: o$.

Recall that for $\mathcal{M}_{4.2.2}$, the initial state is READ.

1. READ $\xrightarrow{\mathsf{r}}$ READ. So for each preparation test case, t, READ $\|_Q (t) =$ READ.
   - $\mathcal{P}_2$ leads to the following test cases and expected outputs:

     $$t = \langle a.a \rangle, h = a, o = \langle a.a \rangle, g = a.$$

2. READ $\xrightarrow{\mathsf{Return}}$ END.
   - $\mathcal{P}_1$ leads to the following test cases and expected outputs:

     $$t = \langle \rangle, h = \mathit{Return}, o = \langle \rangle, g = \varepsilon.$$

   - $\mathcal{P}_4$ is unreachable in state READ.

3. END $\xrightarrow{\mathsf{Return}}$ END.
   - $\mathcal{P}_1$ leads to the following test cases and expected outputs:

     $$t = \langle \mathit{Return} \rangle, h = \mathit{Return}, o = \langle \rangle, g = \varepsilon.$$

   - $\mathcal{P}_3$ leads to the following test cases and expected outputs:

     $$t = \langle a.a.a.a.a \rangle, h = \mathit{Return}, o = \langle a.a.a.a.a \rangle, g = \varepsilon.$$

◇

6.2.1 Faulty transition functions: reprise.

In section 6.2 I discussed faulty transition functions, and how the faults might be revealed. Two concrete coverage measures, the *basic partition-based transition cover* and the *disjoint partition-based transition cover* have been introduced. Both have a formal definition. However, they both have limitations, in that the equivalence partitions they describe are only revealing for faults that affect the sub-functions in a uniform way.

The fundamental problem is that more detailed models of the functions associated with the transitions are required in order to define more detailed fault models (and so to be able to define partitions that reveal more faults). As an open-ended partial solution to this problem, I introduced the notion of a general test-partition generator function, that is intended to embody the fault models of arbitrary testing methods. In this way, I intend that the underlying method of partition-based transition covers can be used with any testing methodology, and can be extended by formally defining test-partition generator functions, based on the structure of the transition functions.

6.3 State identification.

The various coverage sets described are designed to exercise the states and transitions of the implementation with varying degrees of thoroughness. However, once a test case from one of the coverage sets has been used, and given the correct output, it is still necessary to test that it has ended in the expected state.

Definition 6.3.1 For a stream-X-machine $\mathcal{M}$, the set of input sequences, $W \subseteq \Sigma^*$, is a *characterisation set* if

$$\forall\, q_1, q_2 \in Q^{\mathcal{M}} \bullet [\exists\, w \in W \bullet \forall\, m_1 \in \text{Attainable}(q_1);\ \forall\, m_2 \in \text{Attainable}(q_2) \bullet \\ q_1 \mid_\Gamma (m_1, w) \neq q_2 \mid_\Gamma (m_2, w)].$$

◇

Thus, if a characterisation set exists for $\mathcal{S}$, then every pair of states can be distinguished, and so every state can be identified.

Lemma 6.3.2 A characterisation sequence exists for fully-deterministicstream-X-machine $\mathcal{M}$ that is complete w.r.t. Y if $\mathcal{M}$ is Q-Γ-minimal (see definition 4.3.16).

Proof From theorem 4.3.18, $\mathcal{M}$ is Γ-minimal if it is Q-Γ-minimal.

If $\mathcal{M}$ is not Γ-minimal, then a pair of states q_1, q_2 exist that are Γ-equivalent, *i.e.* that respond in the same way to every input sequence, and therefore are indistinguishable. □

Example 6.3.3 In the case of the example, a characterisation set is straightforward:

$\{a\}$

In state Read an a will always be appended to the output, in state End nothing will be appended. ◇

6.4 Extra states.

Extra states can be difficult to detect in a reliable way. Consider the following example:

Example 6.4.1 Consider a specification $\mathcal{S}$, a stream-X-machine of type Φ, with set of states $Q^{\mathcal{S}}$. Suppose it is implemented by $\mathcal{I}$ also of type Φ, but with $Q^{\mathcal{I}} = Q^{\mathcal{S}} \cup \{q'\}$, where $q' \notin Q^{\mathcal{S}}$.

There could be a transition $q \xrightarrow{\phi} q'$ from any of the states $q \in Q^{\mathcal{S}}$, and the particular function ϕ could be defined for arbitrary elements of X, in the same way that any other fault could affect only arbitrary elements. ◇

Some assumptions are needed so that extra state faults can be enumerated.

No extra states. The simplest assumption to make is that there are no extra states, and so there are no faults to reveal. However, this is a very strong assumption, but not necessarily unrealistic in practice.

For instance, in the Petri-net case study of chapter 5, there are nine states, corresponding to the nine different modes of operation of the system. It would require a considerable degree of misunderstanding to add in an extra non-equivalent state for a non-specified extra mode.

One extra state. If there is one extra state, q' say, then there must be a legitimate state, q say, from which it is reachable using a single transition:

$$q \xrightarrow{\phi} q'.$$

A test case of the form $t \frown \langle h \rangle$ will reach state q' iff

$$q_0 \parallel_Q (m_0, t) = q \wedge \Big((q_0 \mid_\Gamma (m_0, t)), (q_0 \parallel_M (m_0, t)), h\Big) \in \text{dom}(\phi).$$

There is no general guarantee that such a $t \frown \langle h \rangle$ exists, since there is no way of knowing what d is for an arbitrary ϕ without knowing the details of ϕ.

However, if the specified machine is fully-deterministic, then ϕ must be defined "at the expense" of some other function, $\overline{\phi}$ say, that would otherwise have been legitimately defined. There are two implications of this:

- $\overline{\phi}$ is faulty in the implementation, since it is not defined as specified.
- Therefore the partition-based transition cover for $\overline{\phi}$ will include tests that result in the coverage of ϕ.

So, as long as the partition-based transition cover in use is good enough, the extra state will be reached.

This does not automatically mean that it will be recognised as an extra state simply by being reached, as ϕ might give the same result as $\overline{\phi}$ would have done if it were not faulty.

Worse still, the characterisation set chosen for $\mathcal{S}$ might not be able to distinguish q' from the specified states.

Example 6.4.2 The characterisation set for $\mathcal{M}_{4.2.2}$ given in example 6.3.3 is not capable of distinguishing arbitrary extra states, for there is only one sequence in the set, $\langle a \rangle$. If, given $\langle a \rangle$ as input, the extra state adds an a to the end of the output sequence, then it would not be distinguished from READ. On the other hand, if the extra state didn't change the output sequence (when given $\langle a \rangle$ as input), it would not be distinguished from END. $\diamond$

The problem is compounded if there might be two or more extra states.

6.5 Test set.

An actual test set, T, will consist of the concatenation of two sets:

- A $\mathcal{G}$ partition-based transition cover, C, where $\mathcal{G}$ is a test-partition generator.
- A characterisation set, $\mathcal{W}$, for the specification.

The sets are concatenated as follows:

$$T = C \cdot \mathcal{W} = \{c \frown w \mid c \in C, w \in \mathcal{W}\}.$$

This will detect all missing state faults, all mis-directed transition faults, and any transition function faults for which C is revealing. It will not detect all extra state faults, as discussed above.

6.6 Test set size.

Given a specification modelled by $\mathcal{S}$, and an associated test-partition generator, what is the upper bound on the size of the test set?

Definition 6.6.1 Suppose that T is a test set. Denote an upper bound on the size of T by $\overline{\uparrow} T$ (*i.e.* size$(T) \leq \overline{\uparrow} T$). ◇

If T is the test set for $\mathcal{S}$, and $T = C \cdot \mathcal{W}$, then

$$\overline{\uparrow} T \leq (\overline{\uparrow} C) \times (\overline{\uparrow} \mathcal{W}).$$

It is quite easy to work out an upper bound for the characterisation set, $\mathcal{W}$, as there needs to be, at most, one element for each pair of states in $\mathcal{S}$. So $\overline{\uparrow}\mathcal{W} = n^2$, where n is the number of states in $\mathcal{S}$.

As for C, suppose that there are f different transition functions, with a maximum of p partitions. So each transition function could need up to *fp* test cases. In the worst case, each transition could be used between every pair of states, once in each direction, *i.e*, up to $n^2 - n$ times. So, $\overline{\uparrow} C = f.p.n^2 - f.p.n$.

Therefore, $\overline{\uparrow} T \leq f.p.n^3(n-1)$.

6.7 Practical applications.

The framework for testing outlined in this chapter has two potential areas of application. Firstly, it can be used as a model for the testing process in general, and secondly, it can be used as a means for the practical generation of test cases.

6.7.1 A general model of testing.

It is worth briefly considering how the general concept of X-machines and the ideas in this chapter could be used as a general model of the testing process.

General X-machines can be used to model almost any system, at a wide variety of levels of abstraction [24]. It is therefore tempting to suggest that any testing can be described in terms of X-machines. As an example, consider trying to model the various levels of structural coverage.

Example 6.7.1 Any piece of code can be modelled as an X-machine. In outline, the pattern is as follows:

1. Each statement in the piece of code corresponds to a state in the model.
2. The transition functions correspond to the semantic functions associated with each statement.
3. The data state (X) corresponds to the memory, registers, buffers, *etc.* of the system running the piece of code.

This model corresponds closely to the traditional idea of a flowgraph.

Each level of structural coverage can be described as a combination of a particular amount of transition or state coverage (most will require full transition coverage), a transition function partition and sometimes a characterisation set. Three of the common levels of coverage are outlined here:—

Statement coverage: Full transition coverage is not required, but full state coverage is required. The transition functions have single partitions (their entire domains).

Branch coverage: Full transition cover is required, but each transition function still has a single partition.

Path coverage: Full transition cover is needed and some of the transition function partitions are complex. Transition functions for "normal" statements have single partitions, but transition functions for "loop" statements are partitioned according to the number of times that the loop has been executed.

No characterisation sequences are needed for these three coverage levels, as such, but each test case has to reach the end of the program. ◇

This is all very well, but the X-machine model being used is not a stream-X-machine, so most of the results in this chapter do not apply.

In fact, general X-machines pose several problems.

1. Firstly, there is no limit to the transition functions that can be used. For instance there is nothing to prevent you using non-computable transition functions.
2. Secondly, the models produced can be non-deterministic, which means that the expected results for test cases cannot be precisely predicted.
3. Thirdly, there are several different ways to minimise an X-machine. In particular, any X-machine is equivalent to a single state X-machine with many complex transition functions. Q-Γ-minimality guarantees that a characterisation exists, but it is only defined for stream-X-machines.
4. Input and output can be handled in a variety of ways.

By defining the concept of stream-X-machines, I overcome these problems to the extent where test sets can always be described, at the cost of limiting the expressive power of the models that can be built.

The model of testing described in this chapter can be applied directly to the testing of any system that can be satisfactorily modelled by a stream-X-machine, by simply devising a suitable partition cover.

6.7.2 A method for test case generation.

Given that a system can be satisfactorily modelled using a stream-X-machine, how practical is it to generate actual test cases using the methods described?

The following steps are certainly needed:

1. Model the specification as a fully-deterministic stream-X-machine that is complete w.r.t. Y and Q-Γ-minimal.

   This stage will require human input, but various kinds of tool support can be envisaged. For instance, it is fairly easy to produce a model that is a stream-X-machine, but it might not be obvious whether the model has all of the other conditions. A tool that analyses X-machines to check for each of the properties would be very helpful.

   - By checking the domains of all the transition functions available in each state in turn, it should be straightforward to check for full-determinism and completeness w.r.t. Y.
   - Unfortunately, checking for Q-Γ-minimality requires that the non-equivalence of compositions of transition functions be checked, and this is not computable, and it is not clear how difficult manual (or machine assisted) proofs of minimality would be in practice.

2. Find a transition cover and a characterisation set.

   Given a stream-X-machine with all the required properties, it is simple to automatically generate a transition cover, since the same techniques that are used for finite-state-machines can be used.

   So long as the stream-X-machine has all of the required properties, a characterisation set exists. Although no guidance on actually generating a suitable set has so far been given, it has proved relatively straightforward in the limited number of case studies so far carried out (such as in the next chapter). However the case studies were carried out manually; it is not known whether characterisation sets can be automatically generated in all cases.

As an alternative, characterisation "features" could be automatically introduced into a stream-X-machine specification, by extending X, all of the transition functions, and the output function so that they record the current state the machine is in. This would make every state immediately recognisable, and so no characterisation set would be required. The drawback of this approach is that the specification has to be artificially extended beyond the actual requirements, but it can be justified by appealing to the concept of "design for test" [13, 15].

3. Choose a suitable test partition generator, $\mathcal{G}$, for the transition functions of the stream-X-machine.

   Depending on the partition cover chosen, and the particular transition functions in use, this stage might be automated. For instance, the two partition based covers proposed in this chapter (basic and disjoint partition based) can be produced automatically by standard partition generator functions, so long as the transition functions are all in the required format.

   Even if the partition cover generator cannot be applied automatically, it only needs to be applied once to each transition function, and so it could be done manually.

4. For each element p of the transition cover (each of which is a path through the machine), find an input sequence for each partition of the last transition function in the Φ-path of p.

   i.e. use $\mathcal{G}$ to produce a $\mathcal{G}$-partition based transition cover for the model of the specification.

   In order to produce a suitable input sequence, it is necessary to use techniques for back propagation from the final expected output to suitable initial input sequences. In cases where all of the transition functions can be inverted (and the inverses are computable), this is straightforward, and can be automated. In cases where some or all of the transition functions do not have computable inverses, it may be possible to find suitable input sequences using various hueristic techniques.

   This is an area of research in itself (see [7]), involving the use of logic programming with various systems of constraints.

5. Use the test cases as normal: Work out the expected output sequence in each case, run the test cases on the implementation, and compare the actual results with the expected results.

   This stage should present no problems for automation.

Chapter 7

Case study: Tests from *X*-machines

In this chapter I apply some of the ideas from chapter 6 to the X-machine model of the Petri-net tool from chapter 5, the specification for which is modelled by stream-X-machine $\mathcal{M}_1$.

But first, here are some of the auxiliary definitions used by the actual transition function definitions.

7.0.3 Auxiliary definitions.

These definitions are repeated from section 5.4.7 (page 65).

$$\begin{aligned}
&\mathit{PSpace} : \mathbb{P}\,\textsc{Positions} \rightarrow \mathbb{P}\,\textsc{Positions}\\
&\qquad = \lambda\, P \bullet \{(x, y) : \textsc{Positions} \mid\\
&\qquad\qquad \exists (x', y') \in P \bullet (x - x')^2 + (y - y')^2 \leq (\mathit{PlaceRadius})^2\}\\
&\mathit{TSpace} : \mathbb{P}\,\textsc{Positions} \rightarrow \mathbb{P}\,\textsc{Positions}\\
&\qquad = \lambda\, P \bullet \{(x, y) : \textsc{Positions} \mid\\
&\qquad\qquad \exists (x', y') \in P \bullet |x - x'| \leq \mathit{TransitionWidth} \wedge\\
&\qquad\qquad\qquad |y - y'| \leq \mathit{TransitionHeight}\}
\end{aligned}$$

where *PlaceRadius* is the radius of a place on the screen, and *TransitionWidth* and *TransitionHeight* are the width and height of a transition on the screen.

$$\mathit{PMiddle} : \textsc{Positions} \times \pi \rightarrow \textsc{Positions} = \lambda (c, P_L) : \textsc{Positions} \times \pi \bullet$$
$$\begin{cases}
\perp & \text{if } c \notin \mathit{PSpace}(\mathrm{dom}(P_L))\\
& \text{ie, if } c \text{ isn't in a place at all}\\
\\
c' & \text{if } c \in \mathit{PSpace}(\mathrm{dom}(P_L))\\
& \text{where: } c' \in \mathrm{dom}(P_L), \text{ and}\\
& \qquad\quad c \in \mathit{PSpace}(\{c'\})
\end{cases}$$

$$\mathit{TMiddle} : \textsc{Positions} \times \tau \to \textsc{Positions} = \lambda(c, T_L) : \textsc{Positions} \times \tau \bullet$$
$$\begin{cases} \perp & \text{if } c \notin \mathit{TSpace}(\mathrm{dom}(T_L)) \\ & \text{ie, if } c \text{ isn't in a transition at all} \\ c' & \text{if } c \in \mathit{TSpace}(\mathrm{dom}(T_L)) \\ & \text{where: } c' \in \mathrm{dom}(T_L), \text{ and} \\ & \qquad c \in \mathit{TSpace}(\{c'\}) \end{cases}$$

$$\mathit{Overlap} = \lambda(\mathit{Points}, \mathcal{L}) : \mathbb{P}\,\textsc{Positions} \times \textsc{Locations} \bullet \begin{cases} \text{true} & \text{if } \mathit{Points} \cap \\ & (\mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \cup \mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau))) \neq \varnothing \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{PlcOverlap} = \lambda(\mathit{Points}, \mathcal{L}) : \mathbb{P}\,\textsc{Positions} \times \textsc{Locations} \bullet \begin{cases} \text{true} & \text{if } \mathit{Points} \cap (\mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \neq \varnothing \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{TransOverlap} = \lambda(\mathit{Points}, \mathcal{L}) : \mathbb{P}\,\textsc{Positions} \times \textsc{Locations} \bullet \begin{cases} \text{true} & \text{if } \mathit{Points} \cap (\mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau)) \neq \varnothing \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{LeftPressed} = \lambda\, m : \textsc{MButtons} \bullet \begin{cases} \text{true} & \text{if } m = \triangleleft \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{RightPressed} = \lambda\, m : \textsc{MButtons} \bullet \begin{cases} \text{true} & \text{if } m = \triangleright \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{NonePressed} = \lambda\, m : \textsc{MButtons} \bullet (\neg\, \mathit{LeftPressed}(m)) \wedge (\neg\, \mathit{RightPressed}(m))$$

$$\mathit{NoKeysPressed} = \lambda\, k : \textsc{Keyboard} \bullet \begin{cases} \text{true} & \text{if } k = \oslash \\ \text{false} & \text{otherwise} \end{cases}$$

7.1 The test-partition generator function for $\mathcal{M}_1$.

The test-partition generator function, $\mathcal{G}_1$, for $\mathcal{M}_1$, is described here on a ϕ-by-ϕ basis, for some of the $\phi \in \Phi$. As the method for constructing the partitions is the same for all ϕ, there is nothing to be gained by producing a complete but very repetitive list of partitions for every one of them.

Instead, consider all the functions used by transitions starting in Place:

$$\mathrm{dom}(F(\textsc{Place})) = \{\mathsf{AddPlace}, \mathsf{DelPlace}, \mathsf{PlaceMode}, \mathsf{Clear}, \mathsf{IgnoreKeyboard}, \mathsf{IgnoreMouse}, \mathsf{TransitionMode}, \mathsf{TokenMode}, \mathsf{NameMode}, \mathsf{ArcMode}, \mathsf{MoveMode}, \mathsf{SaveMode}, \mathsf{LoadMode}, \mathsf{RunMode}\}.$$

7.1.1 $\mathcal{G}_1(\mathsf{AddPlace})$.

AddPlace is the function used to add a place to the net:

$$\mathsf{AddPlace} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), ((c, m, k) :: S)) : X \bullet$$

$$\begin{cases} ((c,\mathcal{L},i,o,\mu) :: G, & \\ \quad((P,T,i,o),\mu,\mathcal{L}),S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \mathit{Overlap}(\mathit{PSpace}\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, left button pressed too close to an existing} \\ & \text{place or transition} \\ ((c,\mathcal{L}',i,o,\mu) :: G, & \\ \quad((P',T,i,o),\mu,\mathcal{L}'),S) & \text{if } \mathit{LeftPressed}(m) \wedge \\ & \neg\ \mathit{Overlap}(\mathit{PSpace}\{c\},\mathcal{L}) \\ & \textit{i.e}\text{, left button pressed away from all the ex-} \\ & \text{isting places and transitions} \\ \bot & \text{otherwise} \end{cases}$$

where,

$$\begin{aligned} \mathcal{L}' &= (pl, \mathcal{L}.\tau, \mathcal{L}.\nu, \mathcal{L}.\delta) \\ P' &= P \cup \{\mathit{newname}\} \\ pl &= \mathcal{L}.\pi \cup \{c \mapsto \mathit{newname}\} \\ \mathit{newname} &\notin P \cup T. \end{aligned}$$

The basic partition of AddPlace is as follows:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge \mathit{Overlap}(\mathit{PSpace}\{c\}, \mathcal{L})\}$.

  This corresponds to the left mouse button being pressed inside an existing place or transition (and away from the menu bar; it doesn't matter whether a key is pressed at the same time).
- $C_2 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge \neg\ \mathit{Overlap}(\mathit{PSpace}\{c\}, \mathcal{L})\}$.

  This corresponds to the left mouse button being pressed away from all the existing places and transitions (and away from the menu bar; it doesn't matter whether a key is pressed at the same time).

The expression $\mathit{Overlap}(\mathit{PSpace}\{c\}, \mathcal{L})$ expands to:

$$\mathit{PSpace}(\{c\}) \cap (\mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \cup (\mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau)))) \neq \varnothing,$$

and, in turn (writing $(c.x, c.y) = c$),

$$\begin{aligned} \mathit{PSpace}(\{c\}) &= \{(x,y) : \textsc{Positions} \mid (x - c.x)^2 + (y - c.y)^2 \leq (\mathit{PlaceRadius})^2\} \\[1em] \mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) &= \{(x,y) : \textsc{Positions} \mid \exists (x', y') \in \mathcal{L}.\pi \bullet \\ &\qquad (x - x')^2 + (y - y')^2 \leq (\mathit{PlaceRadius})^2\} \\[1em] \mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau)) &= \{(x,y) : \textsc{Positions} \mid \exists (x', y') \in \mathcal{L}.\tau \bullet \\ &\qquad |x - x'| \leq \mathit{TransitionWidth} \wedge |y - y'| \leq \mathit{TransitionHeight}\} \end{aligned}$$

The underlying conditions for C_1 and C_2 are very complex and can generate a large number of smaller partitions. To do this a systematic approach to generating them is needed, such as is suggested by Ostrand & Balcer's category-partition method [46] discussed in section 1.4.1. However, in the light of chapter 6, the category-partition method can be formalised slightly, to make its application more precise.

Firstly, simplify the notion of *category* by insisting that a category must correspond precisely to some component element of the type $M \times \Sigma$ (*e.g.* M is a category, and Positions is a category, since it is a component element of Σ). Secondly, the choices for a category must form a partition of it, so that every possible value of the category is represented by precisely one choice for the category.

In principle, the categories for each transition function are different. However, in practice, the functions operate on types that are largely the same as one another, so the categories are also largely the same.

Categories and choices for C_1:

The categories are M, Positions, MButtons, and Keyboard.

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$. The choice is:
   (a) The existing net is not empty (it must have either places or transitions, or both):

   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking of } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$

   where ms is any menu space.

2. The status of the mouse button, m. The only choice is

   $$m = \lhd.$$

   If $m \neq \lhd$ then AddPlace is undefined.

3. The position of the mouse, c. The choices are:
   (a) The mouse is inside an existing place:

   $$c \in \mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)).$$

   (b) The mouse is inside an existing transition:

   $$c \in \mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau)).$$

   (c) The mouse is too close to an existing place, p:

   $$\begin{aligned} &\exists\, p \in \mathrm{dom}(\mathcal{L}.\pi) \bullet \\ &\qquad \mathit{PlaceRadius} < (c.x - p.x)^2 + (c.y - p.y)^2 < 2\mathit{PlaceRadius} \end{aligned}$$

   (d) The mouse is too close to an existing transition, t. There are eight sub-choices: The mouse can be on the left and too close:

   $$\begin{aligned} &\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet \\ &\quad c.x + \mathit{PlaceRadius} > t.x - \mathit{TransitionWidth} \wedge \\ &\qquad |c.y - t.y| < \mathit{TransitionHeight} \end{aligned}$$

   Or too close and on the right:

   $$\begin{aligned} &\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet \\ &\quad c.x - \mathit{PlaceRadius} < t.x + \mathit{TransitionWidth} \wedge \\ &\qquad |c.y - t.y| < \mathit{TransitionHeight} \end{aligned}$$

   Or too close above:

   $$\begin{aligned} &\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet \\ &\quad c.y + \mathit{PlaceRadius} < t.y - \mathit{TransitionHeight} \wedge \\ &\qquad |c.x - t.x| < \mathit{TransitionWidth} \end{aligned}$$

   Or too close below:

   $$\begin{aligned} &\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet \\ &\quad c.y - \mathit{PlaceRadius} < t.y + \mathit{TransitionHeight} \wedge \\ &\qquad |c.x - t.x| < \mathit{TransitionWidth} \end{aligned}$$

Or too close at the top left corner:

$$\begin{aligned}&\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet\\ &\quad (t.x - c.x - \mathit{TransitionWidth})^2 +\\ &\qquad (t.y - c.y + \mathit{TransitionHeight})^2 < \mathit{PlaceRadius}^2\end{aligned}$$

Or too close to the top right corner:

$$\begin{aligned}&\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet\\ &\quad (t.x - c.x + \mathit{TransitionWidth})^2 +\\ &\qquad (t.y - c.y + \mathit{TransitionHeight})^2 < \mathit{PlaceRadius}^2\end{aligned}$$

Or too close to the lower left corner:

$$\begin{aligned}&\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet\\ &\quad (t.x - c.x - \mathit{TransitionWidth})^2 +\\ &\qquad (t.y - c.y - \mathit{TransitionHeight})^2 < \mathit{PlaceRadius}^2\end{aligned}$$

Or too close to the lower right corner:

$$\begin{aligned}&\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet\\ &\quad (t.x - c.x + \mathit{TransitionWidth})^2 +\\ &\qquad (t.y - c.y - \mathit{TransitionHeight})^2 < \mathit{PlaceRadius}^2\end{aligned}$$

(e) The mouse is just too close to an existing place, p:

$$\exists\, p \in \mathrm{dom}(\mathcal{L}.\pi) \bullet (c.x - p.x)^2 + (c.y - p.y)^2 = 2\mathit{PlaceRadius}.$$

(f) The mouse is just too close to an existing transition, t. There are eight sub-choices. The mouse can be to the left of t, to the right of t, above or below t, or at one of the four corners.

$$\begin{aligned}&\exists\, t \in \mathrm{dom}(\mathcal{L}.\tau) \bullet\\ &\quad (t.x \pm \mathit{TransitionWidth} = c.x \mp \mathit{PlaceRadius}) \wedge\\ &\qquad (|c.y - t.y| \leq \mathit{TransitionHeight})\\ &\quad \vee\\ &\quad (t.y \pm \mathit{TransitionHeight} = c.y \mp \mathit{PlaceRadius}) \wedge\\ &\qquad (|c.x - t.x| \leq \mathit{TransitionWidth})\\ &\quad \vee\\ &\quad (t.x - c.x \pm \mathit{TransitionWidth})^2 +\\ &\qquad\quad (t.y - c.y \pm \mathit{TransitionHeight})^2 = \mathit{PlaceRadius}^2\end{aligned}$$

4. The status of the keyboard, k. There are three choices:
   (a) $k = \emptyset$;
   (b) $k \in \{a, \ldots z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

Detailed partitions for C_1:

A very detailed partition of C_1 can be built up by taking one choice from each category to produce each equivalence class of the partition.

Not every combination of choices will correspond to an equivalence class for every instance of AddPlace. For instance, on some occasions, particular choices will involve unattainable Petri-nets.

An example partition is

$$
\begin{aligned}
C_{11} = \{ & \\
& (G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \\
& \quad \mathcal{P} = (P, T, i, o) \wedge \\
& \quad \mu = \text{any marking for } \mathcal{P} \wedge \\
& \quad \mathcal{L} = (\pi, \tau, ms, \langle\rangle) \wedge \\
& \quad m = \triangleleft \wedge \\
& \quad c \in \mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \wedge \\
& \quad k \in \{a, \ldots, z\} \\
\}, &
\end{aligned}
$$

which corresponds to the left mouse button being pressed inside one of the places of a non-empty net at the same time as a key is pressed on the keyboard.

Note Partitions are constructed on the basis that the transition function they are based on could be used anywhere in the machine, even if they are only actually used in a few places.

An example is AddPlace, which is **only** used for the transition

$$\textsc{Place} \xrightarrow{\mathsf{AddPlace}} \textsc{Place}.$$

This means that not all the values described by the partition are attainable. For instance, the only possible value for $\mathcal{L}.ms$ is $\textsc{Pl}$. ◇

Categories and choices for C_2:

1. The existing status of the Petri-net, $G, (\mathcal{P}, \mu, \mathcal{L})$. The choices are:
   (a) The existing net is empty:
   $$\begin{aligned} \mathcal{P} &= (\varnothing, \varnothing, \varnothing, \varnothing) \\ \mu &= \varnothing \\ \mathcal{L} &= (\varnothing, \varnothing, ms, \langle\rangle) \end{aligned}$$
   (b) The existing net is not empty (it must have places, transitions, or both):
   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking of } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$
2. The status of the mouse button, m. The only choice is
   $$m = \triangleleft.$$
   If $m \neq \triangleleft$ then AddPlace is undefined.
3. The position of the mouse, c. The choices are:
   (a) The mouse is away from all the existing places and transitions:
   $$\mathit{PSpace}(\{c\}) \cap (\mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \cup \mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau))) = \varnothing.$$
4. The status of the keyboard, k. There are three choices:
   (a) $k = \oslash$;
   (b) $k \in \{a, \ldots z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

Detailed partition for C_2:

As with C_1, a very detailed partition of C_2 can be built up from the choices of the categories C_2.

7.1.2 $\mathcal{G}_1(\mathsf{DelPlace})$:

DelPlace is the function used to remove a place from the net:

$$\mathsf{DelPlace} = \lambda(G, ((P, T, i, o), \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, i, o, \mu) :: G, & \\ \quad ((P, T, i, o), \mu, \mathcal{L}), S) & \text{if } \mathit{RightPressed}(m) \wedge \\ & \neg\ \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ & i.e, \text{ right button pressed outside all the existing places} \\ ((c, \mathcal{L}', i', o', \mu') :: G, & \\ \quad ((P', T, i', o'), \mu', \mathcal{L}') & \text{if } \mathit{RightPressed}(m) \wedge \\ & \mathit{PlcOverlap}(\{c\}, \mathcal{L}) \\ & i.e, \text{ right button pressed inside an existing place} \\ \perp & \text{otherwise} \end{cases}$$

where

$$\begin{aligned} \mathcal{L}' &= (pl, \mathcal{L}.\tau, \mathcal{L}.\nu, \mathcal{L}.\delta) \\ P' &= P \setminus \{p\} \\ i' &= \lambda\, t : \textsc{Transitions} \bullet (\{p\} \lhd i(t)) \\ o' &= \lambda\, t : \textsc{Transitions} \bullet (\{p\} \lhd o(t)) \\ \mu' &= P' \lhd \mu \\ pl &= \mathcal{L}.\pi \rhd P' \\ p &= P_L(\mathit{PMiddle}(c)) \end{aligned}$$

The basic partition for DelPlace is very similar to that for AddPlace, and is as follows:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid$
  $\mathit{RightPressed}(m) \wedge \neg\ \mathit{PlcOverlap}(\mathit{PSpace}\{c\}, \mathcal{L})\}$.

  This corresponds to the right mouse button being pressed away from all existing places (and away from the menu bar; it doesn't matter whether a key is pressed at the same time).

- $C_2 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid$
  $\mathit{RightPressed}(m) \wedge \mathit{PlcOverlap}(\mathit{PSpace}\{c\}, \mathcal{L})\}$.

  This corresponds to the right mouse button being pressed inside an existing place (and away from the menu bar; it doesn't matter whether a key is pressed at the same time).

As with $\mathcal{G}_1(\mathsf{AddPlace})$, a far more detailed partition can be worked out by considering categories and choices for C_1 and C_2.

Categories and choices for C_1:

1. The existing status of the Petri-net, $G, (\mathcal{P}, \mu, \mathcal{L})$. The choices are:

(a) The existing net is empty:

$$\begin{aligned}\mathcal{P} &= (\varnothing, \varnothing, \varnothing, \varnothing)\\ \mu &= \varnothing\\ \mathcal{L} &= (\varnothing, \varnothing, ms, \langle\rangle)\end{aligned}$$

(b) The existing net is not empty and has both places and transitions:

$$\begin{aligned}\mathcal{P} &= (P, T, i, o)\\ \mu &= \text{a marking for } \mathcal{P}\\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle)\end{aligned}$$

2. The status of the mouse button, m. The only choice is

$$m = \triangleright.$$

If $m \neq \triangleright$ then DelPlace is undefined.

3. The position of the mouse, c. The choices are:

(a) The mouse is away from all the existing places and transitions:

$$c \notin \mathit{PSpace}(\mathrm{dom}(\mathcal{L}.\pi)) \cup \mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau)).$$

(b) The mouse is inside an existing transition:

$$c \in \mathit{TSpace}(\mathrm{dom}(\mathcal{L}.\tau))$$

4. The status of the keyboard, k. There are three choices:

(a) $k = \oslash$;
(b) $k \in \{a, \ldots z\}$, *i.e.* an ordinary character;
(c) $k = \mathit{Return}$.

Categories and choices for C_2:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$. The choice is:

(a) The existing net is not empty and has both places and transitions:

$$\begin{aligned}\mathcal{P} &= (P, T, i, o)\\ \mu &= \text{a marking for } \mathcal{P}\\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle)\end{aligned}$$

2. The status of the mouse button, m. The only choice is

(a) $m = \triangleright$. If $m \neq \triangleright$ then DelPlace is undefined.

3. The position of the mouse, c. The choices are:

(a) The mouse is inside an existing place:

$$\exists\, p \in \mathrm{dom}(\mathcal{L}.\pi) \bullet (p.x - c.x)^2 + (p.y - c.y)^2 < \mathit{PlaceRadius}^2$$

(b) The mouse is on the edge of an existing place:

$$\exists\, p \in \mathrm{dom}(\mathcal{L}.\pi) \bullet (p.x - c.x)^2 + (p.y - c.y)^2 = \mathit{PlaceRadius}^2$$

4. The status of the keyboard, k. There are three choices:

(a) $k = \oslash$;
(b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
(c) $k = \mathit{Return}$.

7.1.3 $\mathcal{G}_1$(PlaceMode).

PlaceMode is the function used to change to the place manipulating state. It does nothing if $\mathcal{M}_1$ is already in state PLACE.

$$\mathsf{PlaceMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet \begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \quad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{PlaceSpace} \\ \bot & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, ms, \mathcal{L}.\delta)$$

The basic partition for PlaceMode is very simple:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{PlaceSpace}.$

  This corresponds to the left mouse button being pressed in the "Place" space of the menu bar.

There is no need to produce quite such a detailed detailed partition for PlaceMode. None-the-less, there is some benefit to be gained from describing categories and partitions for it.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is any net:
   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$
2. The status of the mouse button, m. The only choice is
   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then PlaceMode is undefined.
3. The position of the mouse, c. The only choice is
   (a) The mouse is inside *PlaceSpace*:
   $$c \in \mathit{PlaceSpace}.$$
   PlaceMode is not defined with the mouse in any other positions.
4. The status of the keyboard, k. There are three choices:
   (a) $k = \oslash$;
   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

7.1.4 $\mathcal{G}_1(\mathsf{Clear})$.

Clear is the function used to clear the Petri-net stored in the system, resulting in an empty net.

$$\mathsf{Clear} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}_0, \mathcal{P}_0.\textsc{InArcs}, \\ \quad \mathcal{P}_0.\textsc{OutArcs}, \mu_0) :: G, \\ (\mathcal{P}_0, \mu_0, \mathcal{L}_0), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{ClearSpace} \\ & \textit{i.e.} \text{ if the button is pressed in the ``Clear'' area} \\ & \text{of the menu bar} \\ \bot & \text{otherwise} \end{cases}$$

The basic partition for Clear is very simple:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{ClearSpace}.$

  This corresponds to the left mouse button being pressed in the "Clear" space of the menu bar.

Once again, a more detailed partition can be described using categories and choices.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is already empty:
   $$\begin{aligned} \mathcal{P} &= (\varnothing, \varnothing, \varnothing, \varnothing) \\ \mu &= \varnothing \\ \mathcal{L} &= (\varnothing, \varnothing, ms, \langle\rangle) \end{aligned}$$
   (b) The existing net is any non-empty net:
   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$
2. The status of the mouse button, m. The only choice is
   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then Clear is undefined.
3. The position of the mouse, c. The only choice is
   (a) The mouse is inside *ClearSpace*:
   $$c \in \mathit{ClearSpace}.$$
   Clear is not defined with the mouse in any other positions.
4. The status of the keyboard, k. There are three choices:
   (a) $k = \oslash$;
   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

7.1.5 $\mathcal{G}_1$(IgnoreKeyboard).

IgnoreKeyboard is used to ignore superfluous keyboard input. It should only used when there is no mouse input.

$$\mathsf{IgnoreKeyboard} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, \\ \quad (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{NonePressed}(m) \wedge \neg\ \mathit{NoKeysPressed}(k) \\ \\ \bot & \text{otherwise} \end{cases}$$

The basic partition for IgnoreKeyboard is very simple:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid m = \diamond \wedge k \neq \emptyset\}$
  This corresponds to any key being pressed while no mouse buttons are pressed.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is any net:
   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$
2. The status of the mouse button, m. The only choice is
   (a) $m = \diamond$.
   If $m \neq \diamond$ then IgnoreKeyboard is undefined.
3. The position of the mouse, c. The only choice is
   (a) The mouse is inside anywhere:
   $$c \in \textsc{Positions} \cup \mathit{menubar}.$$
4. The status of the keyboard, k. There are two choices:
   (a) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (b) $k = \mathit{Return}$.

7.1.6 $\mathcal{G}_1$(IgnoreMouse).

IgnoreMouse is used to ignore superfluous mouse input, when no buttons are being pressed.

$$\mathsf{IgnoreMouse} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}, \mathcal{P}.\pi, \mathcal{P}.\tau, \mu) :: G, \\ \quad (\mathcal{P}, \mu, \mathcal{L}), S) & \text{if } \mathit{NonePressed}(m) \wedge \mathit{NoKeysPressed}(k) \wedge \\ & \neg\ \mathit{Overlap}(\{c\}, \mathcal{L}) \\ \\ \bot & \text{otherwise} \end{cases}$$

The basic partition for IgnoreMouse is very simple:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid m = \diamond \wedge k = \emptyset\}$
  This corresponds to no mouse button or keyboard buttons being pressed. The mouse is simply pointing somewhere on the screen.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is any net:
   $$\begin{aligned}\mathcal{P} &= (P, T, i, o)\\ \mu &= \text{a marking for } \mathcal{P}\\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle)\end{aligned}$$
2. The status of the mouse button, m. The only choice is
   (a) $m = \diamond$.

   If $m \neq \diamond$ then IgnoreMouse is undefined.
3. The position of the mouse, c. The only choice is
   (a) The mouse is inside anywhere
   $$c \in \textsc{Positions}.$$
4. The status of the keyboard, k. There is only one choice:
   (a) $k = \emptyset$;

7.1.7 $\mathcal{G}_1$(TransitionMode).

TransitionMode is used to change to the transition manipulation mode.

$$\mathsf{TransitionMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{TransitionSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Tr}, \mathcal{L}.\delta)$$

The basic partition for TransitionMode is:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{TransitionSpace}.$
  This corresponds to the left mouse button being pressed in the "Transition" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is any net:
   $$\begin{aligned}\mathcal{P} &= (P, T, i, o)\\ \mu &= \text{a marking for } \mathcal{P}\\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle)\end{aligned}$$

2. The status of the mouse button, m. The only choice is

   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then $\mathsf{TransitionMode}$ is undefined.

3. The position of the mouse, c. The only choice is

   (a) The mouse is inside *TransitionSpace*:

   $$c \in \mathit{TransitionSpace}.$$

   $\mathsf{TransitionMode}$ is not defined with the mouse in any other positions.

4. The status of the keyboard, k. There are three choices:

   (a) $k = \emptyset$;

   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;

   (c) $k = \mathit{Return}$.

7.1.8 $\mathcal{G}_1(\mathsf{TokenMode})$.

$\mathsf{TokenMode}$ is used to change to the token manipulation mode.

$$\mathsf{TokenMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \quad (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{TokenSpace} \\ \\ \bot & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{To}, \mathcal{L}.\delta)$$

The basic partition for $\mathsf{TokenMode}$ is very simple:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{TokenSpace}$.

  This corresponds to the left mouse button being pressed in the "Token" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.

   (a) The existing net is any net:

   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$

2. The status of the mouse button, m. The only choice is

   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then $\mathsf{TokenMode}$ is undefined.

3. The position of the mouse, c. The only choice is

   (a) The mouse is inside *TokenSpace*:

   $$c \in \mathit{TokenSpace}.$$

   TokenMode is not defined with the mouse in any other positions.

4. The status of the keyboard, k. There are three choices:
   (a) $k = \emptyset$;
   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

7.1.9 $\mathcal{G}_1$(NameMode).

NameMode is used to change to the name manipulation mode.

$$\mathsf{MoveMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \quad (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{NameSpace} \\ \\ \bot & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Na}, \mathcal{L}.\delta)$$

The basic partition for NameMode is:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{NameSpace}$.
  This corresponds to the left mouse button being pressed in the "name" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is any net:

   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$

2. The status of the mouse button, m. The only choice is
   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then NameMode is undefined.

3. The position of the mouse, c. The only choice is
   (a) The mouse is inside *NameSpace*:

   $$c \in \mathit{NameSpace}.$$

   NameMode is not defined with the mouse in any other positions.

4. The status of the keyboard, k. There are three choices:
   (a) $k = \emptyset$;
   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

7.1.10 $\mathcal{G}_1$(ArcMode).

ArcMode is used to change to arc manipulation mode.

$$\mathsf{ArcMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{ArcSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Ar}, \mathcal{L}.\delta)$$

The basic partition for ArcMode is:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{ArcSpace}.$

  This corresponds to the left mouse button being pressed in the "Arc" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.

   (a) The existing net is any net:

   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$

2. The status of the mouse button, m. The only choice is

   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then ArcMode is undefined.

3. The position of the mouse, c. The only choice is

   (a) The mouse is inside $\mathit{ArcSpace}$:

   $$c \in \mathit{ArcSpace}.$$

   ArcMode is not defined with the mouse in any other positions.

4. The status of the keyboard, k. There are three choices:

   (a) $k = \emptyset$;

   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;

   (c) $k = \mathit{Return}$.

7.1.11 $\mathcal{G}_1(\mathsf{MoveMode})$.

MoveMode is used to change to the mode where places and transitions can be moved around.

$$\mathsf{MoveMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \ (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{MoveSpace} \\ \\ \bot & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Mo}, \mathcal{L}.\delta)$$

The basic partition for MoveMode is:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{MoveSpace}$.

  This corresponds to the left mouse button being pressed in the "Move" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.

   (a) The existing net is any net:

   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$

2. The status of the mouse button, m. The only choice is

   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then MoveMode is undefined.

3. The position of the mouse, c. The only choice is

   (a) The mouse is inside *MoveSpace*:

   $$c \in \mathit{MoveSpace}.$$

   MoveMode is not defined with the mouse in any other positions.

4. The status of the keyboard, k. There are three choices:

   (a) $k = \emptyset$;

   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;

   (c) $k = \mathit{Return}$.

7.1.12 $\mathcal{G}_1$(SaveMode).

SaveMode is used to change to the mode for saving nets.

$$\mathsf{SaveMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} \mathsf{Save}(G, (\mathcal{P}, \mu, \mathcal{L}'), (c, m, k) :: S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{SaveSpace} \\ \\ \perp & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Sa}, \mathcal{L}.\delta)$$

The basic partition for SaveMode is:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{SaveSpace}.$
  This corresponds to the left mouse button being pressed in the "Save" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is empty:
   $$\begin{aligned} \mathcal{P} &= (\varnothing, \varnothing, \varnothing, \varnothing) \\ \mu &= \varnothing \\ \mathcal{L} &= (\varnothing, \varnothing, ms, \langle\rangle) \end{aligned}$$
   (b) The existing net is any non-empty net:
   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$
2. The status of the mouse button, m. The only choice is
   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then SaveMode is undefined.
3. The position of the mouse, c. The only choice is
   (a) The mouse is inside *SaveSpace*:
   $$c \in \mathit{SaveSpace}.$$
   SaveMode is not defined with the mouse in any other positions.
4. The status of the keyboard, k. There are three choices:
   (a) $k = \oslash$;
   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

7.1.13 $\mathcal{G}_1(\mathsf{LoadMode})$.

$\mathsf{LoadMode}$ is used to change to the mode for loading new nets.

$$\mathsf{LoadMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} \mathsf{Load}(G, (\mathcal{P}, \mu, \mathcal{L}'), (c, m, k) :: S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{LoadSpace} \\ \bot & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \textsc{Lo}, \mathcal{L}.\delta)$$

The basic partition for $\mathsf{LoadMode}$ is:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{LoadSpace}.$
  This corresponds to the left mouse button being pressed in the "Load" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is any net:
   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$
2. The status of the mouse button, m. The only choice is
   (a) $m = \triangleleft$.

   If $m \neq \triangleleft$ then $\mathsf{LoadMode}$ is undefined.
3. The position of the mouse, c. The only choice is
   (a) The mouse is inside $\mathit{LoadSpace}$:
   $$c \in \mathit{LoadSpace}.$$
   $\mathsf{LoadMode}$ is not defined with the mouse in any other positions.
4. The status of the keyboard, k. There are three choices:
   (a) $k = \emptyset$;
   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

7.1.14 $\mathcal{G}_1(\mathsf{RunMode})$.

$\mathsf{RunMode}$ is used to change to the mode where transitions can be fired to animate the net.

$$\mathsf{RunMode} = \lambda(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \bullet$$
$$\begin{cases} ((c, \mathcal{L}', \mathcal{P}.\textsc{InArcs}, \\ \qquad \mathcal{P}.\textsc{OutArcs}, \mu) :: G, \\ \quad (\mathcal{P}, \mu, \mathcal{L}'), S) & \text{if } \mathit{LeftPressed}(m) \wedge c \in \mathit{RunSpace} \\ \bot & \text{otherwise} \end{cases}$$

where

$$\mathcal{L}' = (\mathcal{L}.\pi, \mathcal{L}.\tau, \text{Ru}, \mathcal{L}.\delta)$$

The basic partition for RunMode is:

- $C_1 = \{(G, (\mathcal{P}, \mu, \mathcal{L}), (c, m, k) :: S) : X \mid \mathit{LeftPressed}(m) \wedge c \in \mathit{RunSpace}.$
  This corresponds to the left mouse button being pressed in the "Run" space of the menu bar.

Categories and choices for C_1:

1. The existing status of the Petri-net, $(\mathcal{P}, \mu, \mathcal{L})$.
   (a) The existing net is any net:
   $$\begin{aligned} \mathcal{P} &= (P, T, i, o) \\ \mu &= \text{a marking for } \mathcal{P} \\ \mathcal{L} &= (\pi, \tau, ms, \langle\rangle) \end{aligned}$$
2. The status of the mouse button, m. The only choice is
   (a) $m = \lhd$.

   If $m \neq \lhd$ then RunMode is undefined.
3. The position of the mouse, c. The only choice is
   (a) The mouse is inside *RunSpace*:
   $$c \in \mathit{RunSpace}.$$
   RunMode is not defined with the mouse in any other positions.
4. The status of the keyboard, k. There are three choices:
   (a) $k = \oslash$;
   (b) $k \in \{a, \ldots, z\}$, *i.e.* an ordinary character;
   (c) $k = \mathit{Return}$.

7.2 Test cases for transitions.

Each basic partition has a large number of detailed partitions. Each detailed partition consists of one choice from each of the categories identified for the basic partition (although not every combination of choices leads to a partition, as some of the choices are incompatible). So, there are very many detailed partitions for each basic partition, and every function has at least one basic partition. There is little to be gained from a detailed discussion of every single one of them.

Instead, consider just $\mathcal{G}_1(\mathsf{DelPlace})$ (section 7.1.2). It is used only once in $\mathcal{M}_1$, for the transition

$$\text{PLACE} \xrightarrow{\mathsf{DelPlace}} \text{PLACE}.$$

It has 2 basic partitions, C_1 and C_2, each of which has four categories, leading to a total of twelve and six combinations of choices for C_1 and C_2 respectively, and therefore to twelve and six test pairs respectively.

One of these pairs corresponds to the choices (1a) "non-empty net with places", (2a) "left mouse button pressed", (3a) "mouse positioned well inside an existing place" and (4a) "no keyboard input" of C_2. Two test cases are required: a preparation test case, $t \in \Sigma^*$, and a transition test case $t \frown \langle h\rangle$ (see definition 6.2.9, page 95).

7.2.1 Preparation test cases:

Firstly choose a value for t such that the machine gets into state PLACE with a non-empty net that has places, bearing in mind the operation of the model, as described in section 5.4.12.

Since PLACE is the initial state, of $\mathcal{M}_1$, this can actually be achieved by a sequence of length one (assuming that the mouse's initial position is in the main screen area):

$$t = \langle(\triangleleft, c_0, \oslash)\rangle.$$

However, this might not be a practical test case. Whilst it might be possible for an automatic test harness to produce such a sequence, a human user would have great difficulty in replicating it. Furthermore, and for the same reason, such an abbreviated test sequence cannot be regarded as typical. Instead something like the following sequence might be a more accurate reflection of the system's actual use (although a human user would have just as much difficulty in reliably producing it precisely).

$$t' = \langle(\diamond, c_0, \oslash), \ldots, (\diamond, c_1, \oslash), (\triangleleft, c_1, \oslash), (\diamond, c_1, \oslash), \ldots, (\diamond, c_2, \oslash)\rangle,$$

where c_2 (the final position of the mouse) is less than a distance of *PlaceRadius* from c_1 (the position of the place added).

If the test cases are to be carried out manually (which would be an immense task), then a more general description of the required preparation sequence would be desirable.

In the case of the example, such a description would simply say

> $(\triangleleft, c_1, \oslash)$ must be included (*i.e.* the left button pressed in position c_1) (to add a new place to the net at position c_1), and that afterwards, there must be no further mouse button presses, and the final element of the sequence must position the mouse within *PlaceRadius* of c_1.

7.2.2 Transition test case:

Having worked out the preparation sequence, this is easy:

$$t \frown \langle(\triangleright, c_1, \oslash)\rangle$$

is the transition test case.

7.3 Characterisation set.

It is possible to recognise each state from the current output (the head of the output stream). This includes a component that describes the state of the menu bar part of the screen, which is different for each state of the machine. So every state is recognisable without needing any additional characterisation sequence.

7.4 Expected output.

The output stream of $\mathcal{M}_1$ is a sequence of values, each of which represents the screen at a particular instant. The head of the output stream represents the latest screen appearance.

A test sequence of length n will lead to a final output stream of length n, and so must have an expected output stream of length of n. In principle, every element of the actual output sequence must be checked against the expected output, and this would be practical for an automated test harness.

In the case of the sequence t' above, the expected output would be

$$\langle(c_0, \mathcal{L}_0, i_0, o_0, \mu_0), \ldots, (c_1, \mathcal{L}_0, i_0, o_0, \mu_0), (c_1, \mathcal{L}_1, i_0, o_0, \mu_0), (c_1, \mathcal{L}_1, i_0, o_0, \mu_0), \ldots$$
$$\ldots, (c_2, \mathcal{L}_1, i_0, o_0, \mu_0)\rangle,$$

where $\mathcal{L}_0, i_0, o_0$ and μ_0 are all initial empty values, and

$$\mathcal{L}_1 = (\{c_1 \mapsto \mathit{place}\}, \tau_0, \mathrm{P{\scriptstyle L}}, \langle\rangle)$$

which is the description of the net with one place (called *place*) positioned at c_1, and no transitions.

If the testing is being carried out manually, a simpler approach is required, as, with many inputs and outputs every second, it would be impossible for the tester to carry out the checks.

Instead, a general description of the output sequence could be used. In the case of t', this would be

> the net remains empty, with the mouse pointer following the movements of the mouse until the left mouse button is pressed at position c_1, when a place is added at the current position of the mouse. The net then remains unchanged, with the mouse pointer following the mouse movements, until finally, it points somewhere within the place at c_1.

Chapter 8

Conclusion

8.1 Testing and formal methods.

There are many benefits to constructing test cases on the basis of a formal model, whether it be of a specification or an implementation. The benefits arise from the ability to precisely describe and reason about potential faults. In particular, it means that tests can be applied uniformly, with greater confidence in their fault detecting potential, and with the possibility of full automation.

There are a great many testing techniques, based on both specifications and implementations. Traditionally, these methods have been described and used informally. Initially this was due to the fact that there were few formal models available to describe specifications, implementations or the potential faults.

However, programs are themselves formal objects, and it is therefore not surprising that program based testing techniques were the earliest developed. More recently formal models and methods for describing specifications have been developed, but they have not been widely used in the field of testing.

8.1.1 Partition testing.

With the case study in chapter 2, I showed how to generate test cases, using Ostrand & Balcer's category-partition method [46], but based on the use of a formal specification in Z. The method is typical of partition techniques.

Although the test cases were comprehensive, and were precisely defined for the individual functions concerned, some limitations were revealed.

- The method does not give any guidance on combining the tests of individual functions into higher level, system wide tests, that ensure each function is tested in all of the different circumstances that are possible.

- The test cases are described in isolation, with no regard for how the particular parameters required for a particular test of a particular function can be produced by using the other system functions. This could be a major problem, if the function to be tested is embedded deep within the system.

- Even using a formal specification, it was by no means clear how to formally describe all the concepts involved. For instance some categories are parameters of the functions, where-as others are particular parts of a parameter. There does not appear to be a general rule. (Although, all of the categories are based, in some way or other, on details of the data-types of the function's input and output parameters.)

8.1.2 Finite-state-machine testing.

In section 3.5, I outlined Chow's W-method [4] for generating test cases from a finite-state machine model of a software specification. The test sets produced are very comprehensive, but not every specification, or aspect of a specification is suited to the finite-state-machine model.

In fact, the aspects suited to finite-state-machine models are largely those that are beyond the partition based functional tests described above, and *vice versa.* A finite-state-machine can easily describe the control structure of a system; the sequences with which low-level functions are used and combined together. But they are not very useful for a detailed description of how those functions behave.

So, the two approaches complement one another well. A model that integrates their good features is needed.

8.2 X-machines.

Eilenberg's X-machine model provides such an integration, by using separate data and control structures. The original model is very general, and, in chapter 4, I introduced various restrictions in order to facilitate testing; so that the model is better suited to modelling interactive systems and to allow models to be refined in a meaningful way. Models conforming to the restrictions are *fully-deterministic stream-*X*-machines that are complete w.r.t. Y.*

8.2.1 Testing X-machines

In chapter 6 I described a test coverage measure for specifications modelled by X-machines that incorporates the features of both finite-state-machine testing and partition based testing. In order to facilitate testing, various extra properties on the X-machines are desirable:

- In order for a characterisation sequence (that distinguishes different states) to exist, the specification must be minimal with respect to its outputs, and the test cases only aim to show that the implementation is equivalent, not that it is minimal.
- The specification must be complete with respect to the possible inputs, so that there is behaviour defined for every possible input in every single state.
- The machine must be fully deterministic, so that there is only one possible output for each input.

Fully-deterministic stream-X-machines that are complete w.r.t. Y possess all of these features.

Each transition is tested once for each of the partitions of the transition function. A formally described simple partition scheme is defined. However, the idea of a partition generator function is introduced, with the intention that the method be open-ended, so that new partitioning techniques can be incorporated.

In the case study (chapter 7) I apply some parts of the method to a large example. By restricting the meaning of a "category" (to component elements of the data-type, where the data-types are Cartesian products of other data-types), I was able to use a formal version of the category-partition method as the partition generator function.

8.2.2 Specification with X-machines.

The method for producing a test set from an X-machine specification is all very well, but the X-machine model needs to be an attractive means for constructing the specification in the first place for the test method to be of practical use. By introducing some very simple formal refinements of the machines, and showing that the refinements preserve properties of stream-X-machines, I hope to have made a start in this direction.

A second influence on the practical use-ability of the X-machine model is the potential for automation.

8.3 Automation of testing.

It is intended that the test cover for X-machines can be generated automatically. Chow outlined a method for automatically producing a transition cover set for a finite-state-machine, and the technique is applicable to X-machines.

What remains is to automatically generate a partition for each transition function. This requires that the transition function be expressed in some suitable form, for which the possible data structures are well understood, and probably restricted. In fact, a functional language, such as Hope or ML would probably be suitable, so long as the functions had to be written in some canonical form and all used the same data-type, and the limitations inherent in executable definitions were acceptable.

An alternative approach would be to semi-automate the process, by allowing the partition generation and test case selection to be directed by a human operator, but most of the repetitive steps to be automated.

8.4 Animation of executable X-machines.

The idea of using a functional language to describe the transition functions suggests that entire X-machines could be executed. In fact it should be possible to produce a tool for writing and animating restricted X-machines. It would be able to check for various useful properties, such as Φ-minimality or completeness w.r.t. Y, and to carry out refinements. It should also be easy for it to convert the machine into the canonical form required for the automated test generation.

8.5 X-machine theory.

The theorems on X-machines that I present in chapter 4 are the bare minimum required to allow the test coverage to be described, and to allow a specification to be refined in a simple and restricted way.

8.5.1 Minimality and equivalence.

Minimisation is not straight-forward with X-machines. For an arbitrary X-machine $\mathcal{M}$, there is an X-machine, $\mathcal{N}$, with only one state and one transition function that exhibits exactly the same behaviour, in terms of the overall input / output function it performs, and so is, in some sense, the minimal version. However, the single transition function would not (necessarily) be of the same type (*i.e.* from the same set Φ) as the transition functions of $\mathcal{M}$, and it would almost certainly be a far more complicated function.

Minimisation of this extreme kind is not of much practical use. The resulting model would be difficult to understand, and would be very unlikely to correspond to any "intuitive" model of the system it was intended to describe.

Instead, I introduce the ideas of Φ-minimal and Γ-minimal, which are based on the minimality within a particular type, Φ. Also, they are defined for stream-X-machines rather than X-machines, which restricts the complexity of each transition function: each one can only deal with a single element of input, *etc.*

8.5.2 Canonical forms of Φ

Having produced a minimal stream-X-machine, it could be described in a canonical form. Two possibilities are:

- Have precisely one transition between each pair of different states, and one "loop-back" transition for each state.

- Have one transition for each possible condition; *i.e*, every transition is of the form

  $$\lambda(G, m, S) \bullet \begin{cases} \phi'(G, m, S) & \text{if } c(m, \mathit{head}\, S) = \text{true} \\ \perp & \text{otherwise} \end{cases}$$

  However, there is the difficulty that the condition function, c, could be decomposed to even simpler condition functions.

8.5.3 Refinements.

The refinements I introduced in section 4.4 were simple and restricted, but are adequate for many purposes, as shown in the case study of chapter 5. In summary, the refinements were based on adding new states, but with a single connecting point to the original machine, and adding new elements to X, but only allowing them to be used in the refined part of the machine. However, more complicated refinements could be defined; for instance:

- Add new elements to X and allow them to be used anywhere in the original machine. This would mean that *all* the transition functions would have to be re-defined, and perhaps α and β too.
- Add new elements to Y as well as to X. This would mean α needed to be re-defined.
- Add new elements to Z, as well as to X. This would mean β needed to be re-defined.
- Add new states so that they connect with the original machine at more than one point.

Each of these would require a careful definition and extra conditions if properties such as fully-deterministic or complete w.r.t. Y were to be maintained by the refined X-machine.

As a refinement of a machine need not be of the same type, it is not generally possible to directly compare the behaviour of a machine with that of a refinement of it. On the other hand, there must (intuitively) be some relationship, involving the behaviour of the refinement *including* the behaviour of the original in some way.

8.6 X-machines at large.

The X-machine model has wider applications than testing.

Firstly and most importantly, it can be used to model entire systems, on many different levels of abstraction.

Using a tool as described above, it would sometimes be possible to rapidly produce an executable version of a specification, and to investigate its properties. The initial specification could then be refined to describe particular features in more detail, until a comprehensive model of the whole system was produced.

The different views of an X-machine model that are possible mean that un-necessary detail can remain hidden much of the time, and make it easier to communicate important features of the model at an intuitive level.

The X-machine model can be used to formally verify properties about algorithms. For instance, in some cases it is possible to enumerate all of the possible paths through an X-machine and compare the corresponding compound input / output functions with a higher level description of the required behaviour. X-machines have also been used to describe the timing constraints of real-time systems.

Beyond these possibilities, the potential for using X-machines in fields such as concurrency or distributed systems has barely been considered.

Index of Notation and Concepts

The notation I use for sets, bags, predicates, functions, relations and sequences is that of the Z specification language, as described in Spivey's reference manual [55].

In addition, I use a large number of concepts, notation and symbols, some from existing literature and some of my own. In the following index, those definitions that are wholly my own, or substantially changed from their traditional meanings, are marked by "*". Items are listed in order of appearance, and the references are to their first appearance and definition.

failure	definition 1.1.2, page 7	
fault	definition 1.1.3, page 7	
affect	definition 1.1.4, page 7	
error	definition 1.1.5, page 7	
correct	definition 1.1.6, page 7	
$\textsc{out}(\_,\_)$	definition 3.1.1, page 32	
$\textsc{ok}(\_)$	definition 3.1.2, page 32	
ideal	definition 3.1.3, page 32	
criterion	definition 3.1.4, page 32	
$\textsc{complete}(\_,\_)$	definition 3.1.5, page 32	
$\textsc{successful}(\_)$	definition 3.1.6, page 32	
$\textsc{reliable}(\_)$	definition 3.1.7, page 33	
$\textsc{valid}(\_)$	definition 3.1.8, page 33	
$\textsc{revealing}(\_,\_)$	definition 3.2.1, page 34	
location	definition 3.3.1, page 34	
fault-based arena	definition 3.3.5, page 35	
alternate sufficient	definition 3.3.8, page 35	
coupling	definition 3.3.9, page 35	
Δ	page 36	*
$\mathit{FaultMap}(\_)$	page 36	*
Finite-state-machine(FSM)	definition 3.4.1, page 37	
(general) behaviour	definition 3.4.1, page 37	
$\lvert\_\rvert$	definition 3.4.1, page 37	
$\lVert\_\rVert$	definition 3.4.1, page 37	
$q \xrightarrow{y/z} q'$	notation 3.4.2, page 37	
	definition 4.1.2, page 43	
complete (FSM)	definition 3.5.1, page 39	
connected (FSM)	definition 3.5.2, page 39	
$\_ \cdot \_$	definition 3.5.3, page 39	
concatenation	definition 3.5.3, page 39	
$\_^{n}$	notation 3.5.4, page 39	
$\_[\_]$	notation 3.5.5, page 39	

$\_ \approx_V \_$	definition 3.5.6, page 39	
V-equivalent	definition 3.5.6, page 39	
$\_ \approx \_$	definition 3.5.7, page 39	
equivalent (FSM)	definitions 3.5.7, 3.5.8, page 39	
minimal (FSM)	definition 3.5.9, page 39	
state cover (FSM)	definition 3.5.10, page 40	
transition cover (FSM)	definition 3.5.11, page 40	
characterisation set (FSM)	definition 3.5.12, page 40	
X-machine(XM)	definition 4.1.1, page 42	
arc	definition 4.1.2, page 43	
path	definition 4.1.3, page 43	
Q-path	definition 4.1.3, page 43	
p_Q	definition 4.1.3, page 43	*
Φ-path	definition 4.1.3, page 43	*
p_Φ	definition 4.1.3, page 43	*
label	definition 4.1.3, page 43	*
$\lvert \_ \rvert$	definition 4.1.3, page 43	*
full-path	definition 4.1.4, page 43	*
partial-path	definition 4.1.4, page 43	*
deterministic (XM)	definition 4.1.5, page 43	*
fully-deterministic (XM)	definition 4.1.6, page 44	*
complete w.r.t. X	definition 4.1.7, page 44	*
complete w.r.t. Φ	definition 4.1.8, page 44	*
stream-X-machine(SXM)	definition 4.2.3, page 47	*
input stream	definitions 4.2.3, 4.2.6 pages 47, 48, 46	*
output stream	definitions 4.2.3, 4.2.6 pages 47, 48, 46	*
memory stream	definitions 4.2.3, 4.2.6 pages 47, 48, 46	*
Output(_)	definition 4.2.6, pages 48	*
Mem(_)	definition 4.2.6, pages 48	*
Input(_)	definition 4.2.6, pages 48	*
Mem_Input(_)	definition 4.2.6, pages 48	*
complete w.r.t. Y	definition 4.2.5, page 48	*
attainable	definition 4.2.7, page 48	*
Attainable(_)	definition 4.2.8, page 48	*
MAttainable(_)	definition 4.2.8, page 48	*
XAttainable(_)	definition 4.2.8, page 48	*
$\_ \mid_\Phi (\_,\_)$	definition 4.3.1, page 49	*
$\_ \mid_\Gamma (\_,\_)$	definition 4.3.1, page 49	*
$\_ \mid_Q (\_,\_)$	definition 4.3.1, page 49	*
$\_ \mid_M (\_,\_)$	definition 4.3.1, page 49	*
$\_ \mid_X (\_,\_)$	definition 4.3.1, page 49	*
$\_ \parallel_\Phi (\_,\_)$	definition 4.3.2, page 50	*
$\_ \parallel_\Gamma (\_,\_)$	definition 4.3.2, page 50	*
$\_ \parallel_Q (\_,\_)$	definition 4.3.2, page 50	*
$\_ \parallel_M (\_,\_)$	definition 4.3.2, page 50	*
$\_ \parallel_X (\_,\_)$	definition 4.3.2, page 50	*
reachable (state)	definition 4.3.6, page 51	*
Φ-equivalent	definitions 4.3.7, 4.3.9 pages 51, 51	*
$\_ \equiv_\Phi \_$	definitions 4.3.7, 4.3.9, pages 51, 51	*
Φ-minimal	definition 4.3.8, page 51	*
Q-minimal	definition 4.3.10, page 51	*
Γ-equivalent	definitions 4.3.13, 4.3.15, pages 53, 53	*

$\_ \equiv_{\Gamma} \_$	definitions 4.3.13, 4.3.15, pages 53, 53	*
Γ-minimal	definition 4.3.14, page 53	*
Q-Γ-minimal	definition 4.3.16, page 53	*
X-machine-let	definition 4.4.1, page 55	*
simple-refinement	definition 4.4.2, page 55	*
general-refinement	definition 4.4.6, page 58	*
Petri-net	definition 5.2.1, page 60	
Places	definition 5.2.1, page 60	
Transitions	definition 5.2.1, page 60	
InArcs	definition 5.2.1, page 60	
OutArcs	definition 5.2.1, page 60	
Marking	definition 5.2.4, page 61	
μ	definition 5.2.4, page 61	
enabled (transition)	definition 5.2.6, page 62	
fire (transition)	definition 5.2.8, page 62	
Locations	page 63	
π	page 63	
τ	page 63	
ν	page 63	
δ	page 63	
Positions	page 64	
MButtons	page 64	
Keyboard	page 64	
$\triangleleft$	page 64	
$\triangleright$	page 64	
$\diamond$	page 64	
$\oslash$	page 64	
$\mathsf{Classes}(\_)$	definition 6.2.2, page 92	*
basic partition-based transition cover	definition 6.2.3, page 92	*
PBTC	definition 6.2.3, page 92	*
disjoint classes	definition 6.2.5, page 94	*
$\mathsf{DClasses}$	definition 6.2.5, page 94	*
disjoint partition-based transition cover	definition 6.2.6, page 95	*
test partition generator	definition 6.2.7, page 95	*
recognises	definition 6.2.8, page 95	*
partition-based test	definition 6.2.9, page 95	*
(un) reachable (value)	definition 6.2.9, page 95	*
preparation test case	definition 6.2.9, page 95	*
transition test case	definition 6.2.9, page 95	*
$\mathcal{G}$-PBTC	definition 6.2.10, page 95	*
$\overline{\uparrow}$	definition 6.6.1, page 99	*

Bibliography

[1] J. R. Abrial. *B-Tool Reference Manual.* BP International Limited and Edinburgh Portable Compilers Ltd, BP Innovation Centre, Slough, 1991. Version 1.1.

[2] BCS Specialist Interest Group in Software Testing. *A Standard for Software Component Testing*, November 1990. Issue 1.2, edited by Dorothy Graham and Martyn Ould.

[3] Asok Bhattacharyya. *Checking Experiments in Sequential Machines.* Wiley Eastern Limited, New Delhi, India, 1989.

[4] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[5] O. J. Connelly. Graphical Petri-net editor. Final year project report, Department of Computer Science, University of Sheffield, June 1991.

[6] Richard A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[7] Mark Dunn. Integrating hardware testing and design—a role for constraint satisfaction. Project Report 3, University of Sheffield, Department of Computer Science, Sheffield, September 1991. DTI/SERC sponsored IEATP program, project IED2/1/1031: *Functional Testing for High Integrity Systems.*

[8] M. Dyer and A. Kouchakdjian. Correctness verification: Alternative to structural software testing. *Information and Software Technology*, 32(1):53–59, January/February 1990.

[9] Samuel Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.

[10] Samuel Eilenberg. *Automata, Languages and Machines*, volume B. Academic Press, New York, 1974.

[11] W. R. Elmendorf. Functional analysis using cause-effect graphs. In *Proceedings of SHARE XLIII*, New York, 1974. SHARE.

[12] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[13] David Gelperin and Bill Hetzel. The growth of software testing. *Communications of the ACM*, 31(6):687–695, 1988.

[14] Susan L. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, pages 7–10, September 1990.

[15] Christopher Paul Gerrard, Derek Coleman, and Robin M. Gallimore. Formal specification and design time testing. *IEEE Transactions on Software Engineering*, 16(1):1–11, January 1990.

[16] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI–CSL–88–9, SRI International, Computer Science Laboratory, California, USA, August 1988.

[17] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[18] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[19] Pat Hall, Tom Gedeon, and Chris Reade. Formal methods, testing, and reuse—towards reliability conservation for software. In *Colloquium on Software Testing for Critical Systems* [29]. Organised by Professional Group C1 (Software Engineering) of the IEE. Digest No.: 1990/108. Chairman: Darrel C. Ince.

[20] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[21] Ian J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, January 1986.

[22] Bill Hetzel. *The Complete Guide to Software Testing.* QED Information Sciences, Wellesley, Massachusetts, first edition, 1984.

[23] Mike Holcombe. *Algebraic Automata Theory*, volume 1 of *Studies in Advanced Mathematics.* Cambridge University Press, Cambridge, 1982.

[24] Mike Holcombe. X-machines as a basis for dynamic system specification. *Software Engineering Journal*, March 1988.

[25] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3), September 1976.

[26] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[27] William E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, New York, 1987.

[28] William E. Howden. Program testing versus proofs of correctness. *Journal of Software Testing, Verification and Reliability*, 1(1):5–15, April–June 1991.

[29] IEE. *Colloquium on Software Testing for Critical Systems*, Savoy Place, London, 19th June 1990. Organised by Professional Group C1 (Software Engineering) of the IEE. Digest No.: 1990/108. Chairman: Darrel C. Ince.

[30] IEEE. *Standard Glossary for Software Engineering Terminology*, 1983. IEEE Standard 729–1983.

[31] Pankaj Jalote. Testing the completeness of specifications. *IEEE Transactions on Software Engineering*, 15(5), May 1989.

[32] Donald Ervin Knuth. *The TEXbook.* Addison-Wesley, New York, 1984.

[33] Leslie Lamport. *LATEX: A Document Preparation System.* Addison-Wesley, New York, 1986.

[34] Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Digest of Papers: 15th International Symposium on Fault-Tolerant Computing*, pages 2–11. IEEE, June 1985.

[35] Gilbert Laycock. Formal specification and testing: A case study. *Software Testing, Verification and Reliability*, 2(1):7–23, 1992.

[36] Gilbert Laycock and Mike Stannett. X-machine workshop. Research Report CS–92–08, University of Sheffield, Department of Computer Science, Sheffield, 1992.

[37] Keith W. Miller, Larry J. Morell, Robert E. Noonan, Stephen K. Park, David M. Nicol, Branson W. Murrill, and Jeffrey M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, January 1992.

[38] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[39] Carroll Morgan. *Programming From Specifications.* Series in Computer Science. Prentice Hall International, London, 1990.

[40] Glenford J. Myers. *Software Reliability.* J. Wiley and Sons, New York, 1976.

[41] Glenford J. Myers. A controlled experiment in program testing and code walkthroughs/-inspections. *Communications of the ACM*, 21(9):760–768, September 1978.

[42] Glenford J. Myers. *The Art of Software Testing.* J. Wiley and Sons, New York, 1979.

[43] National Bureau of Standards, Washington D.C. *Guideline for Lifecycle Validation, Verification and Testing of Computer Software*, 1983. Report NBS FIPS 101.

[44] NATO/MoD. *Interim Defence Standard 00–55, Requirements for the Procurement of Safety Critical Software in Defence Equipment*, draft edition, 1989.

[45] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–873, June 1988.

[46] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[47] Nigel Perry. *Hope*$^+$. Functional Programming Research Group, Computer Science Department, Imperial College, London, February 1988. Issue 5, IC/FPR/LANG/2.5.1/7.

[48] James Lyle Peterson. *Petri-Net Theory and Modelling of Systems.* Prentice-Hall, Inc, 1981.

[49] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall International, London, 1991.

[50] Wolfgang Reisig. *Petri-Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[51] György E. Révész. *Lambda-Calculus, Combinators, and Functional Programming*, volume 4 of *Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1988.

[52] Marc Roper and P. Smith. A structural testing method for JSP designed programs. *Software—Practice and Experience*, 17(2):135–157, February 1987.

[53] Marc Roper and P. Smith. A specification-based functional testing method for JSP designed programs. *Information and Software Technology*, 30(2):89–98, March 1988.

[54] Deepinder P. Sidhu and Ting-Kau Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426, April 1989.

[55] J. Michael Spivey. *The Z Notation, A Reference Manual.* Series in Computer Science. Prentice Hall International, London, first edition, 1989.

[56] Steven Vickers. *Topology Via Logic*, volume 5 of *Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1989.

[57] S. N. Weiss and Elaine J. Weyuker. An extended domain-based model of software reliability. *IEEE Transactions on Software Engineering*, 14(10):1512–1524, October 1988.

[58] Elaine J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, June 1988.

[59] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

[60] Martin R. Woodward. Mutation testing—an evolving technique. In *Colloquium on Software Testing for Critical Systems* [29]. Organised by Professional Group C1 (Software Engineering) of the IEE. Digest No.: 1990/108. Chairman: Darrel C. Ince.

[61] D. Wu, M. A. Hennell, D. Hedley, and I. J. Riddell. A practical method for software quality control via program mutation. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 159–170, Banff, Canada, July 1988. IEEE.