NTNU
Norwegian University of
Science and Technology

Compiling Regular Expressions into Non-Deterministic State Machines for Simulation in SystemC

Kjetil Volden

Master of Science in Electronics
Submission date: August 2011
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Project description

Based on the class library developed in a project for functional simulation of non-deterministic state machines, the following tasks will be performed:

- to investigate principles for using regular expressions in system descriptions, and develop a grammar for including such expressions in SystemC,
- and to define a method for generating functionally equivalent nondeterministic finite state machine descriptions from regular expressions in SystemC.

Abstract

With Moore's law exponentially increasing the number of transistors on integrated circuits, developers fail to keep up. This makes chip area an increasingly cheap resource. At the same time, researchers and developers are trying to find ways to dynamically reconfigure FPGAs, preferably at run time, so as to increase the flexibility of hardware solutions, and close the gap between the speed of hardware and flexibility of software. A proposed way of solving both of these issues at once is by using nondeterministic finite-state machines as a fundamental unit of design. This could provide great flexibility and dynamic hardware solutions, but before this can be known for sure, a system like this would need to be simulated. This paper documents the planning and development of a SystemC library that creates nondeterministic finite-state machines from regular expressions, and a special regular expression syntax designed for this specific application. The paper can also be used as a reference for the inner workings of, and how to use, the library.

Contents

Chapter 1

Introduction

Since the dawn of electronics, and the integrated circuit particularly, we have witnessed an incredible development in information processing. The circuits are getting faster, more dense and more complex, and use less power, at a rate that exceeds any other area in technology. Moore's law predicts that the number of transistors that can be inexpensively placed on an integrated circuit will roughly double every two years. This idea was originally presented in a paper by Gordon Moore in 1965 where it was calculated to double every year[1], but he later refined the calculations to show a doubling every two years.[2] Designers are struggling to keep up, and this results in what is commonly known as the 'productivity gap'. The designers are simply not able to create systems that utilize all the available area, and the number of transistors a design team can make use of is increasing at a slower rate than Moore's law. This means that chip area is becoming an increasingly cheap resource.

Simultaneously, some designers are trying to get around a related issue. Tra-

ditionally, custom tailored hardware solutions on FPGAs and ASICs are used when there is a need for very efficient processing of the same task many times over. Examples are coders/decoders, information encryption, and hardware acceleration of computer graphics. And when there is a need for a general purpose system that can do practically all kinds of different tasks, serial processing of changeable program instructions are often used. Examples are microcontrollers and personal computers. In other words, custom tailored hardware solutions give efficiency where serial processing gives flexibility. But what if it would be possible to make a custom tailored hardware solution that would change according to the task that needed to be performed? Enter *dynamically reconfigurable systems*.

These systems, to my knowledge, do not currently exist outside of research labs, but as the name suggests, these are hardware solutions that can be changed dynamically. For instance, an OpenGL hardware accelerator could be redefined into a Deep Blue chess computer in a matter of moments. This would introduce a great deal of flexibility without necessarily loosing a lot of efficiency. Many of these proposed systems need to be reconfigured from outside of the system itself, but not all. And some even aim for a dynamically reconfigurable system that can reconfigure parts of the circuit while running. These are called *runtime reconfigurable systems*.

Recently, my supervisor, professor Kjetil Svarstad at the Norwegian University of Science and Technology, proposed using self cloning nondeterministic finite-state machines (NDFSM) as a fundamental unit of design for such runtime reconfigurable systems.[3] However, to prove that this can be done it is necessary to simulate such systems, and earlier this year I completed a project that made this possible. It was a nondeterministic finite-state machine library

in SystemC which was able to perform tasks assigned to states in the form of callback functions.[4] This master's thesis is about using that concept as a basis for designing a more complex system that allows for hardware description through the use of regular expressions.

1.1 Motivation

The concept of runtime reconfigurable systems is an attempt at closing the gap between software and hardware, by fusing the flexibility of software with the speed and efficiency of hardware. One way to go about this is to use dynamic elements as fundamental building blocks in hardware, where these change or duplicate according to some input. Nondeterministic finite-state machines are such dynamic elements, and by making such state machines that clone themselves when facing several state transitions from the same state, one could possibly achieve a great increase in flexibility and processing speed at the cost of chip area. In principle, one would implement solutions that contain descriptions of *potential* hardware blocks that would be realized on chip in real time as they are needed.

In order to prove that this would in fact work, and to more clearly define the boundaries of what a system like this can and cannot do, the ability to simulate this is an absolute necessity. I have already made a simplistic solution for this, as already mentioned, but this had some limitations as well as being quite tedious to define—especially for more complex systems. It could only take one input, could not handle "NOT-transitions" or ranges in input, the state machine had to be described in detail with all states and transitions explicitly defined, and it used function pointers to class methods as a callback system for performing

tasks associated with states—all in all a limited and verbose solution, but a solution nonetheless.

Therefore, we decided to improve and expand on this concept to make something similar, but without the verbosity and with fewer limitations. It has been proven that if something can be described with nondeterministic finite-state machines, it can also be described with regular expressions—as well as the other way around. In other words, for every nondeterministic (and deterministic) finite-state machine there is a functionally equivalent regular expression, and this means that we can use regular expressions to define the state machines that in turn defines a dynamic hardware description. Defining dynamic hardware solutions with regular expressions could prove to be both powerful and quick, but to prove this we need a working model for simulation purposes. A limitation of SystemC is that modules can not be created or destroyed while in simulation[5, p. 11], so the model can not be self cloning, but instead needs to be able to contain several active states at once—however, this should be sufficient as a proof of concept.

The system should also accept several inputs, and the input type should not be restricted more than absolutely necessary. Consequently, a custom tailored adaptation of regular expression syntax and semantics will need to be defined.

1.2 Report structure

The literature surrounding the concept that is explored and developed throughout this task is sparse. There are a few articles about how nondeterministic finite-state machines can be used to solve timing issues on FPGAs and other integrated circuits, or how regular expressions can be compiled into PLUs, and

similar topics. But, generally speaking, they are either outdated, only superficially touching the same areas as this, or both. Because of this, I cannot write much more about the history of this concept or bring about pages of references and citations.

This is more or less a new idea, and the work has been consisting of a large amount of thinking, and much less doing. This is not a report chuck full of simulations, tests and empirical data, but a documentation of the process of developing the programmatic foundation of something that *will be* simulated, tested and producing empirical data. Throughout the report I have tried to explain why I have made the decisions I have made and what alternatives were considered wherever this is not self evident.

My contributions to this project are this report, the development of a specific adaptation of regular expression into a syntax suitable for this specific application, an algorithm for translating a regular expression of this kind and syntax into a nondeterministic finite-state machine of equivalent functionality, the development of the complete SystemC library described in this report, and designing and coding the examples of practical use of the library.

The project was given to me and started on the 12th of May 2011, and finished and handed in 13 weeks later on the 12th of August 2011.

I decided to split this report into five main chapters in addition to the abstract, introduction, discussion and conclusion, and I have also tried to structure the report in a way that makes it usable as a reference for the actual SystemC library that has been developed. After this introduction, you will find a chapter about nondeterministic finite-state machines. This chapter describes what they are, what separates them from deterministic finite-state machines, how they are defined formally, and a part about a special type of transitions, before ending

the chapter with a short part about state machine actions.

Then follows a chapter about regular expressions. First I explain a bit about regular expressions in general, and especially the parts relevant to this task, and then I explain in detail the whats, hows and whys about the specific adaptation of regular expression syntaxes that was developed and defined for this application. The chapter ends with a few examples of regular expressions using this adapted regular expression syntax.

The next chapter goes on to explain how one can translate a regular expression to a nondeterministic finite-state machine. A fairly well known algorithm for doing this, known as *Thompson's algorithm*, is described, and I continue with explaining why I did not use that as it is, and describe a somewhat similar algorithm that I developed as part of this project that is a little more fitting with the specifications and limitations this application demands.

In the following chapter, the programmatic implementation in SystemC is described and explained in detail. The chapter starts with a few general observations and notes, before I walk the reader through a couple of new C++ features that are parts of the new and upcoming 2011 standard for C++, and that I have taken advantage of. Following this is a section about how the system is structured programmatically, before I go through all the classes with their members and methods, as well as a few global functions, and explain what the system does, how it does it, and why.

The next chapter contains three separate and different examples of how to use this SystemC model, and they all describe different aspects of using it. Hopefully this will both show some of the features of the library and give the reader some idea of how this can be used, and what it can be used for, in addition to being used on the detailing level of fundamental design units.

Before going on to the conclusion, there is a discussion chapter where I discuss a few ideas of what could be improved, what could be expanded, and what could be added to make the library better and easier to use.

After the conclusion follows an appendix containing the complete code of the SystemC library. This is separated in five sections—one for each header file. And at the very end of the document, the list of code listings, list of figures, list of tables and the bibliography can be found.

Chapter 2

Nondeterministic finite-state machines

A master's thesis should as far as reasonably possible be self contained. Thus, the report would be lacking without a chapter on nondeterministic finite-state machines. Having already written most of this in my previous project, I fail to see the point in rewriting the entire chapter in different words, so I have chosen to include it in an *almost* verbatim fashion. More specifically I can say that the rest of this introduction to nondeterministic finite-state machines, counting from the next paragraph, as well as Chapter 2.2, which explains epsilon-moves, and Chapter 2.1, which explains NDFSM formalism, are included almost verbatim, and with only minor changes. Chapter 2.3 is written specifically for this report, and explains the essentials of actions associated with states and transitions.

A nondeterministic finite-state machine is a finite-state machine where for each pair of state and input symbol there may be several possible next states,

as opposed to a deterministic finite-state machine where the pairs of states and input symbols are uniquely determined. This can be interpreted as a state machine where you can't know which of these possible next states will be active until the input symbol has been consumed and the transition has occurred—with or without probability weighting of the possible transitions, as is the case in the more general concept of *probabilistic automata*—or as a state machine which can have more than one active state at the same time. In this paper, and the programming model, I will use the latter interpretation, as it opens for the possibility of using such an NDFSM to implement parallelism in a hardware setting.

Nondeterministic finite-state machines do not carry greater computational power than deterministic finite-state machines (DFSM), as it is always possible to convert any NDFSM to an equivalent DFSM that recognizes the same formal language through a method called *powerset construction*[6]. However, the equivalent DFSM can often require far more transitions and states than the original NDFSM, and this relationship can potentially be exponential; if an NDFSM has n states, the equivalent DFSM can require up to 2^n states. Because of this, NDFSMs can often be much more flexible, and often easier and less tedious to design, than an equivalent DFSM.

Another aspect of NDFSMs is that one doesn't have to specify every possible input symbol as transitions from every state. A trivial example of this is shown in Figure 2.1. If an active state has no specified transition for a given input, the state will simply transition into nothing, leaving the state inactive—unless, of course, a preceding state immediately reactivates it through a normal transition.

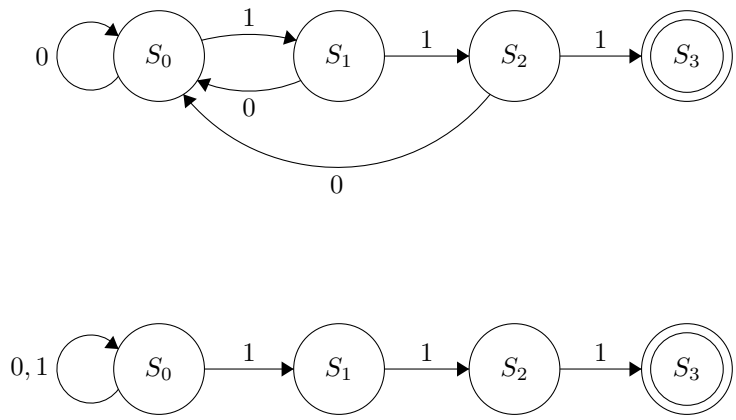

Figure 2.1 – An example of the difference between a DFSM (top) and an NDFSM (bottom), where both are made to enter the accept state S_3 after the first occurrence of three consecutive ones in the input string.

2.1 NDFSM formalism

The NDFSM formalism do not differ much from regular DFSM formalism. There are, however, two different kinds of NDFSMs with regards to formal definitions—regular NDFSMs, and NDFSMs with ϵ-*moves* (or simply *epsilon-moves*). The regular NDFSM is defined by the 5-tuple (Q, Σ, T, q_0, F), where:

- Q is a finite set of states,
- Σ is a finite set of input symbols called the input alphabet,
- T is a transition function, $T : Q \times \Sigma \to P(Q)$,
- q_0 is a start state, and
- F is a set of accept states, and a subset of Q ($F \subseteq Q$).

Here, $P(Q)$ denotes a power set of Q, and this is what differs from the definition of DFSMs. In DFSM formalism, $T : Q \times \Sigma \to P(Q)$ is replaced by

	0	**1**
$\boldsymbol{S_0}$	S_0	S_1
$\boldsymbol{S_1}$	S_0	S_2
$\boldsymbol{S_2}$	S_0	S_3

(a) δ (DFSM)

	0	**1**
$\boldsymbol{S_0}$	$\{S_0\}$	$\{S_0, S_1\}$
$\boldsymbol{S_1}$	$\{\ \}$	$\{S_2\}$
$\boldsymbol{S_2}$	$\{\ \}$	$\{S_3\}$

(b) T (NDFSM)

Table 2.1 – The state transition tables of the FSMs in Figure 2.1.

$\delta : Q \times \Sigma \to Q$ – the difference being that the former defines a transition to a set of possible states, whereas the latter defines a transition to one uniquely determined state.

Using the NDFSM in Figure 2.1 as an example, the values of the 5-tuple would be: $Q = \{S_0, S_1, S_2, S_3\}$, $\Sigma = \{0, 1\}$, $q_0 = S_0$, $F = \{S_3\}$, and T would be defined by the state transition table shown in Table 2.1. Also shown, to illustrate the difference between the definition of a DFSM and an NDFSM, is the state transition table that defines δ of the DFSM in the same figure.

2.2 Epsilon-moves

NDFSMs can be defined with a kind of extension, allowing transitions to occur regardless of input. Such a state machine is often referred to as an *NFA-ϵ*, an *NFA-λ*, or simply an *NFA/NDFSM with epsilon-moves*. The proper definition of an epsilon-move, or epsilon transition, is that it does not consume an input symbol, so such a transition can occur even if the input is an empty string. This means that the transition function T must be redefined to

	0	**1**	ϵ
$\boldsymbol{S_0}$	$\{\ \}$	$\{S_1\}$	$\{S_0\}$
$\boldsymbol{S_1}$	$\{\ \}$	$\{S_2\}$	$\{\ \}$
$\boldsymbol{S_2}$	$\{\ \}$	$\{S_3\}$	$\{\ \}$

Table 2.2 – A state transition table with ϵ-moves.

$T : Q \times (\Sigma \cup \{\epsilon\}) \to P(Q)$

Here, ϵ represents a zero-length string, and can as such be thought to be anywhere in the input string. A transition table for a state machine with epsilon-moves follows the same principles as for those without, but with ϵ as an additional input symbol. An example is shown in Table 2.2.

2.3 Actions

The states of the state machine can have actions associated with them. An action is an activity that is scheduled to be performed at a given moment. In general there are four different types of actions:

- **Entry action**: performed when entering the state.
- **Exit action**: performed when exiting the state.
- **Input action**: performed dependent on present state and input conditions.
- **Transition action**: performed when performing a certain transition.

Readers familiar with finite-state machines will know that Moore machines use entry actions, and Mealy machines and combinatorial machines—finite-state machines with only one state—use input actions. The implementation done as

part of this thesis uses entry actions, but this is covered in greater detail in later chapters.

Chapter 3

Regular expressions

I have already pointed out that regular expressions and nondeterministic finite-state machines are functionally equivalent, and I have gone through the details of what defines a nondeterministic finite-state machine and how they work. In this chapter I start by going through some of the theory behind regular expressions, and continue with explaining how ordinary regular expression syntax and semantics have been adapted to this specific project. The latter part also contains a table that lists and explains all the metacharacters that are used in this adaptation.

3.1 Regular expressions theory

Regular expressions sprung out of the computer sciences of automata theory and formal language theory. Both of these fields deal with computation models, and a precursor to regular expressions, called *regular sets* was a mathematical

notation invented by Stephen Cole Kleen in the 1950s used to describe such computation models.[7] This notation was later implemented in the editor QED by the famous computer scientist Ken Thompson—who also designed a fairly well known algorithm for translating regular expressions into nondeterministic finite-state machines[9], which I will describe in a little more detail in a later chapter—to enable pattern matching in text files. This was the birth of regular expressions, and regular expressions as a means for pattern matching in strings and text files was later included in many text editors.

Traditionally, a regular expression provides a way to match strings of text. There are a number of different syntaxes, like POSIX Basic Regular Expressions, POSIX Extended Regular Expressions and Perl regular expressions, where some provide more functionality than others, but they all work by the same general principles. A regular expression defines a set of possible strings that are compared to another string—as such one could say that regular expressions provide a means to defining a large set of alternative strings for matching purposes without having to list all the elements of the set. It is considered a *match* if the string to be compared is identical to one or more of the strings in the set defined by the regular expression. Usually in practice, when doing pattern matching, a longer string is scanned for parts that match the regular expression, so when matching a string it suffices if only part of the string matches one or more elements in the string set defined by the regular expression.

Most syntaxes provide the same basic operations for constructing regular expressions. These are the boolean OR-operator, grouping or subexpressions, and quantification.

The boolean OR-operator takes the form of a vertical bar and is used to separate different alternatives in the expression. For example, the expression

ab|cd matches the strings *ab* and *cd*. Note that these two strings could be part of a larger string, and don't need to define the whole string to be matched in its entirety. This fact is to be taken implicitly throughout the chapter unless otherwise explicitly specified. Grouping, or creating subexpressions, is done by enclosing the subexpression in parentheses, like in the expression *(ab|cd)ef*, which would match the strings *abef* and *cdef*. Subexpressions can be nested.

Quantifiers succeed the character or subexpression they are associated with, and define how many consecutive repetitions of this element are allowed to occur. The three basic quantifiers are the question mark (?), the asterisk (*) and the plus sign (+). These imply *zero or one*, *zero or more* and *one or more*, respectively. Many syntaxes also include a different kind of quantification that explicitly defines a minimum and maximum number of allowed repetitions—in POSIX Extended Regular Expressions, for instance, this is written as $\{m,n\}$, where m is the minimum and n is the maximum.

Some syntaxes, like the Perl syntax and Perl-derivatives, include a few more concepts like *lazy quantification*, *case insensitivity*, *named capture groups* and *recursive patterns*,[8] but none of these concepts are especially relevant to this task, so I will not go into this in more detail.

The characters in a regular expression that serve some special purpose are called *metacharacters*. Quantifiers, parenthesis and the vertical bar are all metacharacters, and the number of metacharacters vary between the different syntaxes. However, in most syntaxes, but not all, metacharacters are escaped—meaning that they are interpreted as normal characters—by preceding them with a backslash.

Most syntaxes also have a wildcard metacharacter that matches any character that may occupy the placeholder it refers to. This metacharacter is usually

the dot character (.). For instance, the expression $a.b$ will match any three character string starting with an a and ending with a b.

Square brackets are usually used as a way to list several alternatives to one character position. An example could be *[abc]* which would match the characters a, b or c, and be another way of writing $a|b|c$. Square brackets also allow *ranges*. Ranges are defined by putting a hyphen between two characters that mark the opposite ends of the allowed range of characters. For instance *[a-z]*, which would accept any lowercase character between a and z inclusive. The square brackets can contain several ranges, as well as a mix of ranges and single values. An example that matches any alphanumeric character and the underscore character could be *[A-Za-z0-9_]*. If the contents of the square brackets start with a caret, the results are inverted, meaning a character matches if it is *not* in the values and ranges defined—for example *[^A-Za-z0-9]*, which would match any non-alphanumeric character.

Some syntaxes also have shorthands for expressing certain things like alphanumeric characters, word boundary characters, digits and so on—usually referred to as *escape sequences*—but how this is expressed syntactically varies. Almost all syntaxes also have metacharacters to define an association with the very start and end of the whole string to be matched against as well, but this is not something that will be used in this project, as the start and end of input strings is a concept often rather diffuse, if not meaningless, in this context, and can be solved through other means if somehow necessary.

3.2 Adaptation of syntax and semantics

The regular expression syntax used in this model is based on *POSIX Extended Regular Expressions*, but contains a few modifications to enable matching against several input signals and different input types. In ordinary regular expressions, each character is either a metacharacter dictating some functionality or a character to match some string of characters against. For our purposes, on the other hand, several strings of values will sometimes be evaluated in a parallel and interdependent fashion, and the values will not necessarily be characters—they can just as well be integers, floating point numbers, boolean values, bit vectors or character strings, to mention a few possibilities.

Allowing different data types demand that all tokens, or units of evaluation, need to be syntactically separated to avoid ambiguity surrounding where one unit ends and the next begins. Allowing several inputs also sets a need for grouping values that correspond to different inputs together with a boolean relationship between these values. There is also a need for specifying references to actions that will be associated with states at different positions in the expressions. I will go through these details below.

The values to be checked against must either be written in a way that corresponds to how the values are written and read through C++ streams using the stream extraction and stream insertion operators, or the explicit template specialization of the casting functions in Chapter 5.8. Insertion to and extraction from streams form the functionality of the general function, while strings, c-type strings and characters have explicit template specialization to allow whitespace characters. If this does not fit ones needs, one would either have to overload the data type's stream extraction and insertion operators, or define ones own

explicit template specialization of the casting function.

Subexpressions follow the same pattern as for ordinary regular expressions. That involves the use of parentheses, and use of the vertical bar (|) as an OR-operator. Subexpressions can be quantified with the use of the designated quantifier metacharacters. Escaping any character with a backslash forces it to be interpreted as a regular character.

3.2.1 Unit of evaluation

The specifications mentioned above demands the definition of some unit of evaluation. This unit contains the values to be evaluated and the relationship between them within a set of angle brackets. An example of a unit for evaluating one character input to 'a' would look like this: <a>. This will simply be referred to as a *unit*. The angle brackets were chosen because they are the only easily accessible type of brackets that are not already reserved as metacharacters serving a different function.

When evaluating several inputs, the values of these inputs are separated by a metacharacter describing their boolean relationship—either '&', denoting a boolean *AND*, or '|', denoting a boolean *OR*. An example of a unit for evaluating one integer input to 10 *and* another integer input to 14 would look like this: <10&14>. The parts of a unit separated by boolean operators—which hold the values of 10 and 14 in the previous example—will be referred to as *sub-units*. The number of sub-units per unit has to be consistent throughout the regular expression, but the different sub-units within a unit do not have to be of the same data type. The signs for boolean functions were chosen by convention.

A sub-unit can be written as nothing more than a single value to matched against, or as several possibilities grouped in square brackets. In ordinary regu-

lar expressions, values in square brackets are written next to each other without whitespace or punctuation, which is fine when the input data type is limited to single characters. This would cause problems in our case, as the input is not limited to single characters, or single digits. Therefore, these values are separated by commas, and their placeholders will be referred to as *fields*—the group of fields will be referred to as a *field set*. A field can be a single value, like the fields in the field set [3,4,5], or a range. A range is a set of two values separated by a hyphen, and it is considered to be a match when the input is between the two values inclusive. In other words, the field set [3,4,5] is equivalent to the field set [3-5]. A field set can consist of both single values and ranges—for instance [1,3-5,12], which would match the inputs 1, 3, 4, 5 and 12. It needs to be noted, however, that ranges are only valid for data types that can actually be compared with the C++ <= and >= operators. Also note that whitespace characters like space, tabulator and newline are accepted and treated literally in the regular expressions—although escape sequences might be introduced in the future.

If the first character in the field set is a caret (^), this implies that the field set is inversed. That means that it is considered to be a match when the input does *not* match the fields it contains. This can be thought of as inverting the boolean value of the result of matching against a field set without the caret, but which is otherwise identical.

Units can be quantified with the designated quantifier metacharacters. Units can also be made with different data types as input—integers and characters, for instance—and as usual, boolean *AND* has higher precedence than boolean *OR* when combinations of the two are used.

3.2.2 Actions

Since these regular expressions will be translated into a nondeterministic finite-state machine with associated actions, there must also be a way to specify these actions in the regular expression. Actions are specified with an @-sign succeeded by an integer, and are placed after, and outside of, the units they apply to, and after the unit's quantifier, if there is one. The action is associated with the state that will be activated when the transition whose condition is defined by the preceding unit is performed. The integer succeeding the @-sign refers to a zero-based index of a vector or array of functors that is passed as a separate argument to the system. An @-sign without a succeeding integer is interpreted as referring to index zero. This can be practical when operating with a single functor.

The @-sign was chosen simply because, semantically, it traditionally functions as the preposition *at*, referring to a syntactically succeeding position—in emails it points to a domain holding the incoming email server, and here it points to an index of functors. I did consider dropping the index and interpret the expressions with increasing index from left to right, but this both complicates the expression parsing and can necessitate defining the same functor several times in the vector or array of functors, as well as making it difficult to use the same vector or array of functors for several pattern matching modules with different regular expressions.

3.2.3 Metacharacters

The metacharacters and their functionality are listed in Table 3.2.3.

Metacharacter	Description
<>	Encapsulates a unit for evaluation, as explained in Section 3.2.1.
&	Boolean AND-operator that denotes the relationship between two inputs, i.e. *<a&b>*, which matches the case where the first input is 'a' and the second input is 'b'.
|	This has two functions. 1) A boolean OR-operator between sub-units; and 2) a boolean OR-operator between the expression before it and the expression after it, i.e. *<a><b>|<c><d>*, which matches the character streams {a, b} or {c, d}.
( )	Defines a subexpression. For example, *<c>(<a><t>|<o><d>)* matches the character streams {c, a, t} or {c, o, d}.
{*m*, *n*}	Matches the preceding element at least *m* times, and at most *n* times. For example, *<a>{1,3}* matches the streams {a}, {a, a} and {a, a, a}.

continued on next page...

...continued from previous page

Metacharacter	**Description**
+	Matches the preceding element one or more times.
*	Matches the preceding element zero or more times.
?	Matches the preceding element zero or one time.
[ ]	Placed inside a unit, it matches one of the elements inside. Contains a field set, which can consist of one or more fields. The fields are separated by commas, and ranges can be denoted by using hyphens for types that support *greater than or equal to* and *less than or equal to* checks. For example, *<c&[1,10-100]>* matches the case where the first input is 'c' and the second input is either 1 or between 10 and 100 inclusive.

continued on next page...

...continued from previous page

Metacharacter	**Description**
[^]	Exactly the same as the '[ ]' metacharacters, except it starts with '[^' and means anything *except* the elements inside.
,	Separates the fields of a field set.
-	Denotes a range for types that support less-than-or-equal and greater-than-or-equal checks. See the description of the '[ ]' metacharacters for an example.
.	When used alone, and not as a field value in a field set, or a part of a character string, floating point value or something similar, it is a wildcard that matches anything. For example, *<a&.>* matches the case where the first input is 'a' no matter what the second input is. If it is not used alone, it is treated as a regular character and not a metacharacter. For example, *<a&3.14>* matches the case where the first input is 'a' and the second input is the floating point or string value of 3.14.

continued on next page...

...continued from previous page

Metacharacter	Description
@i	Marks where an action will be performed, where i is an integer representing a functor array or vector index. The integer can be omitted, which is equivalent to writing @0. These are explained in more detail in Chapters 3.2.2, 2.3 and 5.
\	Escape character used for interpreting reserved metacharacters as regular characters. For example, <\&> matches the character '&'.

Table 3.1 – Metacharacters of regular expressions.

3.2.4 Some examples

At this point it is assumed that the reader is familiar with regular expressions in general, as well as the specific adaptations mentioned in the previous chapters. Therefore, the following examples will only have a short literal descriptions and explanations without going into the syntactic details.

<1&1><1|1>(<0&0>|<1&1>) :
Two inputs. First, both must be 1; second, one or both must be 1; third, either both must be 0 or both must be 1.

<S><T><A><R><T>(<0 >@0|<1 >@1|<[2-9]>@2)@3 :
One input. The first five incoming inputs must be the characters S, T, A, R and T. Then the input must be either 0, which would trigger the functor with index 0, or it must be 1, which would trigger the functor with index 1, or it must be between 2 and 9 inclusive, which would trigger the functor with index 2. If the subexpression for the last input is a match, meaning the input is between 0 and 9 inclusive, functor with index 3 would be triggered.

<0|0|0>(<1&1&1>|<1&0&0>|<0&1&0>|<0&0&1>) :
Three inputs. The first is a three input *NAND*, and the following subexpression is a three input *XOR*.

Chapter 4

From regular expression to NDFSM

From the fact that regular expressions and nondeterministic finite-state machines are functionally equivalent, it follows that it is possible to translate one into the other. The most famous algorithm for doing this is arguably *Thompson's algorithm*. This was first described in an article Ken Thompson wrote in 1968, where he describes how to compile regular expressions for an IBM 7094 computer using the language ALGOL-60. This algorithm is of historical importance, and is more or less the conceptual foundation of my own algorithm, so I will start by explaining the foundations of Thompson's algorithm before I go on to explain my own, and why I chose not to use Thompson's algorithm instead.

4.1 Thompson's algorithm

Thompson's algorithm was made to compile regular expressions to machine code, so that it could be used for very fast matching of text strings. The resulting code would take the text to be matched as input, and find all substrings in the text that matched the regular expression.

This compiler consisted in three concurrently running stages, where the first stage would examine the regular expression to make sure that it is not malformed. The second stage would reorganize the whole regular expression into reverse Polish form, with a special juxtaposition operator, symbolized with the character '$\cdot$'. To use the same example as Thompson, the expression *a(b|c)*d* would be translated to *abc|*$\cdot$d$\cdot$* in reverse Polish form. The next step would be to introduce a pushdown stack, and start pushing the characters onto the stack, where the elements on the stack are pointers to the compiled code of an operand. When the boolean OR-operator (|) or the juxtaposition operator ($\cdot$) is encountered, the two topmost elements on the stack are combined, and a new pointer to this operation replaces the two elements. The result is thereby available as an operand to another operation. Because the '|' and '$\cdot$' operators operate on two values, they are called binary operators. When a quantifier, like '*', is encountered, this acts as a unary operator which only acts upon the topmost element on the stack. The quantifiers have defined operations that replace the last element with an operand of the functionality represented by the quantifier. Finally, when the last character or operator of the regular expression has been compiled, the pushdown stack holds only one element, and that is the pointer to the code for the whole regular expression.

Using the same example as above, the process would begin by pushing the

elements a, b and c on the stack in that order. Then the '|' operator would replace the two topmost elements with a pointer to the compiled code of the operand $b|c$. Then the '*' quantifier would replace the topmost element with a special construct looping the $b|c$ element back around itself to a search path splitting node, so that zero or more occurrences of b or c would lead to the same place in the search path. Next up is the '$\cdot$' operator that combines the two topmost elements on the stack to one by attaching them to each other, search path wise. Then d is pushed onto the stack, and the last '$\cdot$' combines the two. The stack is now left with one pointer, pointing to the code for the whole regular expression.

4.2 My own algorithm

Although Thompson's algorithm is a very efficient algorithm for compiling ordinary regular expressions into machine code, it does not use states in the same way this system will have to use states. Instead it uses two kinds of nodes called NNODEs and CNODEs that represent characters and search path splitters respectively. Using states is essential for this project because the states represent actual hardware modules with functionality. Also, using functor markers in the regular expressions both makes this syntactically more complicated with Thompson's algorithm, and this state machine might demand more states since two parallel path ways might be set to trigger different functors. I therefore decided to make a new algorithm with Thompson's algorithm as a foundation. But instead of first translating the regular expression to reverse Polish form, I decided to make a recursive algorithm instead.

Before describing this recursive algorithm in a step-by-step fashion, I would

like to point out a couple of things. First I will discuss something I call the *fundamental building blocks* that are used as a basis when a unit or subexpression is quantified with either no quantifier, the *zero or more* quantifier (*), the *one or more* quantifier (+), or the *zero or one* quantifier (?). These are all solved by creating a new state and making transitions between the states by a certain pattern. These patterns can be seen in Figure 4.1, where they are shown with a simple unit as the transition conditions. However—and this is very important—these fundamental building blocks are also used for subexpressions. In other words, a fundamental building block is not just two states with simple and single transitions between them, but patterns of connection that correspond to the different quantifiers. So a quantified subexpression would be connected between these two states replacing the single unit in the figure, with any epsilon transitions exactly as shown.

The other thing I would like to point out is that units or subexpressions quantified with the {*m,n*} quantifier are equivalent to a subexpression of *n-m+1* parallel alternatives, where the first alternative is the unit or subexpression repeated *m* times, the second alternative is the unit or subexpression repeated *m+1* times, and so on up to the last alternative which is the unit or subexpression repeated *n* times. For instance, the quantified unit <*a*>{*2,4*} is equivalent to the expression <*a*><*a*>|<*a*><*a*><*a*>|<*a*><*a*><*a*><*a*>.

The translation from regular expressions to nondeterministic finite-state machines in a system that supports actions will differ somewhat depending on what type of action is used. This system will perform actions when certain input conditions are met, which leaves the options of transition actions and entry actions. I chose entry actions because they seemed easier to implement, especially when using the fundamental building blocks explained above as a basis.

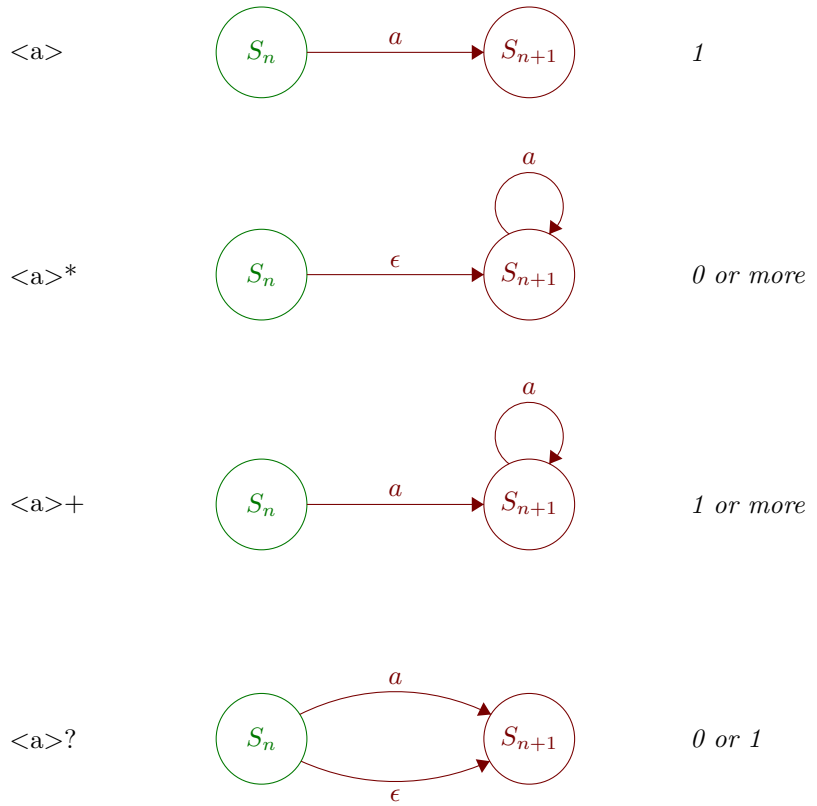

Figure 4.1 – The fundamental building blocks of regular expression to NDFSM construction. Left: regular expression units with quantifiers; center: NDFSM fundamental building blocks; right: literal description. The units can be exchanged with complete subexpressions, with epsilon transitions still connected as shown.

In the following step-by-step description of the algorithm, there are four values referred to as *start-state*, *end-state*, *old-state* and *new-state*. When assigning values to them, this is implicitly done by reference, so states are not copied. The first two are incoming arguments, and the last two are local variables used to keep track of the states as they are created and transition conditions are defined. The first call to this recursive function happens after the first state has been created, and this is passed as the *start-state* argument, while *end-state* is *NULL*. In the description, *token* refers to a either a unit with succeeding quantifier and functor declaration, if any, a subexpression with succeeding quantifier and functor declaration, if any, or a boolean OR-operator.

1. Set *old-state* to *start-state*.

2. Try to read the next token.

3a. If the token is a unit or subexpression quantified with the $\{m,n\}$ quantifier, translate it to a new equivalent subexpression and go to 3c.

3b. If the token is a unit, create a new state and set *new-state* to this, create transitions between *old-state* and *new-state* according to the fundamental building blocks, and associate appropriate functor to *new-state* if any. Go to 2.

3c. If the token is a subexpression, create a new state and set *new-state* to this, call self with subexpression as new regular expression, *old-state* and *new-state* as *start-state* and *end-state* arguments according to the fundamental building blocks, and associate appropriate functor to *new-state* if any. Go to 2.

3d. If the token is a boolean OR-operator, save *old-state* in a local stack, and set *old-state* to *start-state*. Go to 2.

3e. If no token could be read because of reaching the end of the string, create

epsilon transitions from *old-state* and all states in the local stack to *end-state* if *end-state* is not *NULL*.

When the recursive loop is finished, the complete regular expression is translated into a nondeterministic finite-state machine, with functors associated with the states representing the search path positions at which their respective functors were declared in the regular expression.

Chapter 5

SystemC implementation

In this chapter I will describe how the SystemC implementation works, how it is designed and why it is designed the way it is. I have divided the chapter into several sections, where the first describes and explains some new C++ features that are used in the implementation, the second explains the programmatic structure of the system, and the rest describes each part of the structure in detail. The complete code is listed in the appendix and can be a useful resource when reading the following sections, but in addition, every section that explains a class or a list of functions will contain an overview of the relevant members, methods and/or functions.

When designing the library, care was taken to make the model easy to use. Declaring, defining, initializing and using the model should not be complicated, tedious or overly verbose. It should have as little limitations as reasonably possible when it comes to accepted data types of the inputs, as well as the number of inputs.

5.1 C++11 (C++0x)

Since this is a future oriented project, I decided to take use of some of the new and upcoming C++ functionality. A new standard for C++ has been in the works for quite some years, and was originally supposed to be released in 2008 or 2009. For this reason, the standard was, and is, commonly referred to as *C++0x*, but is being finalized this year, and it is now often referred to as *C++11*. The additions and changes are fairly numerous and I have no intention of naming them all here, but I will briefly explain the two new concepts that are used in this project. To prevent any unnecessary confusion, I must also point out that C++11 allows for juxtapositioning of right angle brackets in template declarations, like *vector<a_class<int>>*, something its predecessors C++98 and C++03 did not, and that I have used this syntax consistently in the project's code.

Also note that to compile a program with C++11 functionality with the compilers of today, the compiler must be run with the appropriate command line parameters. For GCC (G++), the compiler argument is "*-std=c++0x*", or "*-std=gnu++0x*" to enable GNU extensions—the latter is necessary for the SystemC library to compile.

5.1.1 Lambda functions

Lambda functions are function objects—also known as functors—with certain characteristics and a special syntax. They can operate on variables in their parent scopes both by reference and by copy, they can take arguments and return values. They are very practical for using as functors that perform actions associated with the states, and are the default value for the template parameters

of the system. Lambda functions follow the following syntax.

```
[capture_mode](parameters) mutable throw() -> return_type {body}
```

Listing 5.1 – C++11 lambda function syntax.

The *[capture_mode]* dictates how variables from the surrounding scope when the lambda was defined (not when they are called) are captured. They can be captured by reference, by value, or not at all. An ampersand means that variables are captured by reference; an equals sign means that the variables are captured by value; and an empty set of brackets means that variables are not captured at all. However, one can also specify which variables are to be captured by reference, which variables are to be captured by value, and which variables are not to be captured. Some examples follow.

```
[]       // Variables are not captured
[&]      // All variables are captured by reference
[=]      // All variables are captured by value
[a, &b] // a is captured by value, b by reference
[a, &]  // a is captured by value, all others by reference
[&a, =] // a is captured by reference, all others by value
```

Listing 5.2 – C++11 lambda function capture modes.

The *(parameters)* section behaves like the parameters section of other function definitions, except for three things: 1) The parameter list cannot have default arguments, 2) the parameter list cannot have a variable length parameter list, and 3) the parameter list cannot have unnamed parameters. However, it can be omitted if, and only if, the function has no parameters *and* the return type is implicit *and* there is no exception specification—I will get back to what an implicit return type means shortly.

The keyword *mutable* needs to be included if the function changes the value of a variable captured by value—otherwise an error will be thrown. This is because variables captured by value are considered to be *consts* in the functions' scope. Note that the variable does not change value externally, even if it is changed inside the lambda function with the mutable keyword, as long as it is captured by value.

Lambda functions can have exception specification like ordinary functions. This specification comes between the *mutable* keyword and the return type, and is optional.

Return types are written with a preceding hyphen and right angle bracket, similar to the arrow operator, followed by the data type that the function returns. Explicitly stating the return type can be omitted if the return type is void, or if the function body is a single line in the form *return expression;*. If omitted, the function's return type is said to be implicit.

The {*body*} section is, of course, the function body, and contains C++ code just like ordinary functions.

The data type of a lambda function is *function<return_type (parameter_type_1, parameter_type_2, ...)>*. This means that lambda functions with different return types or different parameters cannot be elements of the same array, vector or similar container structure. In other situations it is rarely necessary to explicitly state the data type of the lambda function, as they are often defined and used in ways that do not demand them being stored as variables, and because the *auto* keyword has been extended in C++11 as a shorthand for declaring any variable type that is known at compile time—a concept called *type inference*. Examples of both cases follow.

```
#include <iostream>
```

```
#include <functional>

using namespace std;

int main(int argc, char* argv[]){
  int a = 3, b = 5;
  int c = [=]{return a + b;}();
  auto l = [&](int d)->float{++a; return (float)(c)/d;};
  cout << a << " " << c << " " << l(3) << endl;
  return 0;
}
```

Listing 5.3 – Trivial examples of defining and using C++11 lambda functions.

Compiling and running the code above would result in the output "*4 8 2.66667*".

In addition to being much less tedious to define than self defined function objects, a small part of the reason I chose to introduce lambda functions in the design is that some kind of template argument would have to be supplied when declaring the PMatch object to specify the functor data type, and the ability to default this argument to something practical and predefined opens for somewhat less verbose code on the user end—*PMatch<> pmatch*, versus *PMatch <My_own_functor> pmatch*.

Lambda functions are supported by GCC version 4.5 and higher[10], and Visual C++ 2010 and higher.[11]

5.1.2 Variadic templates

It has been possible, in both C++ and C, to have functions that are able to take a variable amount of arguments through the *cstdarg* library. This makes use of several macros to enable the unpacking and looping through the arguments that are passed.

In C++11, a new concept called variadic templates is introduced. This allows for templates to take any amount of arguments, which can be used for both class and function definitions. The syntax is fairly similar to the *cstdarg* syntax, but does not allow for looping through the arguments. Instead, recursion is commonly used. The syntax is shown in a very simple example below:

```
#include <iostream>

using namespace std;

template<typename T>
void print(T first){
  cout << first << endl;
}

template<typename T, typename... Args>
void print(T first, Args... rest){
  cout << first << endl;
  print(rest...);
}

int main(int argc, char* argv[]){
  print(12, 'f', "string", 3.14, true);
  return 0;
}
```

Listing 5.4 – Simple example showing how C++11 variadic templates can be used.

There are two meanings to the ellipsis operator. When it occurs on the left of the parameter name, it declares a *parameter pack*. A parameter pack can be used in a function's parameter list to represent zero or more arguments. When the ellipsis occurs on the right of the parameter name, it *unpacks* the arguments, and they can be used as arguments in another function call—usually one that also implements variadic templates, and often itself in a recursive manner. In the example above, the *print()* function prints the first argument and calls itself with the rest of the arguments. This will go on recursively until the parameter pack is empty, in which case it calls an overloaded version that takes only one argument. The one-argument version prints the last argument and ends the recursive loop. This ends up printing the arguments from right to left in the parameter order, but to change the order one could simply define the print function as *void print(Args... rest, T last)* instead—where the parameter names are completely arbitrary, of course.

I have used variadic templates in the methods that are involved in stepping the state machine with inputs, so that the data types and the number of inputs are not limited. However, using variadic templates was a more difficult decision, because they are not supported in Visual C++ 2010. Because of this, I considered using the *cstdarg* library or overloading the methods that use this functionality with varieties that take from one to a reasonable high amount of templated arguments. Some research revealed that the next version of Visual C++, commonly referred to as *Dev11*, will support variadic templates[12]. This method of creating variadic functions and methods will most likely be the stan-

dard in the future, and the *cstdarg* will equally likely become deprecated at some point. At the same time, compiling on Windows is still possible today using the Windows port of GCC, called *MinGW*. With this is mind, I decided to use variadic templates instead of the alternatives.

Variadic templates are supported by GCC version 4.3 and higher[10].

5.2 Functors

A functor is a function object, and is traditionally implemented as a class with its parenthesis operator overloaded so that it can be called as a function. PMatch uses lambda functions as the default functors, but user defined functors can also be used.

One of the easier ways to implement user defined functors where different functors of the same type are to do different things without being called with arguments, is to define a class that has a variable that is used within its overloaded parenthesis operator function body to decide what functionality should be executed. This variable can be passed as a constructor argument, so that the functor is initialized with a certain "active" functionality. A trick for making the functor operate outside its scope is to define pointers of the data type of variables, signals, et cetera, that are to be acted upon by the functors, and to pass the addresses of these variables and signals as constructor arguments—or through other methods, of course.

A couple of things need to be kept in mind when using user defined functors. One is that the functor needs to have a parameterless constructor defined, even if it has an empty body. That is because the states contain functors, and when initializing the states, the parameterless constructor of the functor will be called.

The other thing is that some functors may have members of complex data types, and since associating a functor with a state involves copying the state through the assignment operator, this might need to be overloaded in certain cases. C++ handles this automatically in most cases.

It would save memory and make any possible need for overloading of the assignment operator unnecessary if I had chosen to use functor pointers in the states instead. There are several reasons why I did not choose this. For one, it could cause unexpected behavior if the functor objects were deleted, changed or moved in memory during simulation. Second, the states are supposed to represent actual hardware modules, and as such they should hold their own functionality.

The class name, or data type, of the functor is passed as a template argument when declaring the PMatch object.

An example showing how user defined functors can be defined and used is shown and explained in Chapter 6.3.

5.3 System structure

The system is structured in a hierarchical way, and is in many ways similar to the hierarchical structure I have used to explain the regular expressions, with units, sub-units, field sets, fields, and ranges and values. The main class is called *PMatch*. PMatch holds several *State* objects, which represent the states of the state machine. State objects can hold *Rule* objects, which correspond to regular expression units, and Rule objects hold *Sub_rule* objects that correspond to sub-units and field sets in the regular expression syntax. The Sub_unit objects holds the values and ranges that correspond to the regular expression fields. Only the

PMatch class inherits *sc_module*. It could be said that it would be appropriate for the State class to also inherit *sc_module*, as the states actually represent hardware modules, but since *sc_module* objects cannot be created or deleted during simulation, this would make it impossible to initialize and re-initialize a PMatch object after the simulation has started.

I have deliberately avoided using ordinary arrays in the code, and use vectors instead—except for the container of State objects, which is a deque, but this is explained later. Vectors are much more convenient than arrays, in that they are dynamic and have boundary checking when accessing elements.

All methods that are not supposed to be accessed from outside of the class in question are private, and the same goes for all the class members.

The next chapters explain the different classes, their methods and their members in detail.

5.4 The PMatch class

The PMatch class is the top class. This is the class one instantiates objects of or inherits when using the system. It contains only one member, a *deque* of states, but a range of private and public methods. An overview of the class can be seen below.

```
template <class Functor=std::function<void()>>
class PMatch : public sc_module{
private:
  std::deque<State<Functor>> states;

  int max(int a, int b);
  int get_highest_callback_index(std::string reg_exp);
```

```
  std::string extract_subexpression(std::string expression, int
      start_pos);
  std::string extract_unit(std::string expression, int start_pos);
  std::string build_quantified_subexpression(std::string expression
      , int min, int max);
  void build_from_expression(std::string reg_exp, State<Functor> *
      start_state, State<Functor> *end_state, std::vector<Functor>&
       functions);

public:
  PMatch(sc_module_name name);
  void initialize(std::string reg_exp, Functor *functor, bool
      continuous = true);
  void initialize(std::string reg_exp, Functor functor, bool
      continuous = true);
  void initialize(std::string reg_exp, std::vector<Functor>
      functions, bool continuous = true);

  template <typename... Inputs>
  void step(Inputs... inputs);

  void restart();
  bool is_empty();

  #ifdef PMATCH_DEBUG
  void debug_output_structure();
  #endif
};
```

Listing 5.5 – An overview of the members and methods of the PMatch class.

The reason for using a deque, and not a vector, as a container for the states is that elements in a deque maintain their position in memory for the lifetime of the deque, while elements in a vector are not guaranteed to keep their memory address when elements are added or removed. The states activate each other through pointers, so maintaining memory location is completely necessary. However, it would be possible to have the states activate each other via the PMatch class using indexes instead of pointers, but this would be a much more complicated solution.

PMatch takes a template argument that defines the functor object that is to be used. This defaults to lambda functions with return type void and no parameters, as lambda functions are, in my opinion, far easier to use, and much less tedious to define, than self defined function objects. Having a well chosen default template argument also makes the code less verbose on the user end.

When instantiated, PMatch remains empty of states. It is initialized with the *initialize(...)* function, which takes the regular expression as the first argument. The method is overloaded, so that it takes either a single functor, an array of functors or a vector of functors as the second argument. The third argument is a boolean value that controls whether the resulting state machine's first state will perform a conditionless self transition for every step—the argument is optional, and defaults to *true*, meaning that it does perform self transitions. In other words, the last argument controls whether the state machine will run continuously, or if it has to be restarted when an input symbol results in all states being deactivated.

On initialization, PMatch builds the state machine according to the previously described algorithm. This involves creating the states, and using the states' methods to create the transition conditions, all according the aforemen-

tioned algorithm.

I will go through the methods from top to bottom in the order they are written in the overview above—skipping the constructor, as it does absolutely nothing more than pass the *sc_module_name* to the parent class *sc_module*.

5.4.1 int max(...)

This method takes two integer arguments and returns the largest one. This is done in a straight forward manner and should require no further explanation.

5.4.2 int get_highest_callback_index(...)

When PMatch is initialized with an array of functors, it actually just receives the pointer to the first functor in the array. This method is then used to scan through the regular expression to find @-signs that are not escaped, and extract and return the highest succeeding integer.

5.4.3 std::string extract_subexpression(...)

This method takes two arguments—a string and an integer. The first is a regular expression that contains a subexpression, and the second is the start position of the subexpression to be extracted, including the opening parenthesis. The method keeps track of any additional non-escaped opening parentheses it encounters and returns the subexpression without the enclosing parentheses.

5.4.4 std::string extract_unit(...)

Working in roughly the same way as the method immediately above, this takes the regular expression as its first argument, and start position of a unit as the

other. It returns the unit *with* the enclosing angle brackets.

5.4.5 std::string build_quantified_subexpression(...)

A part of the regular expression syntax is the special quantifier that specifies the minimum and maximum amount of repetitions. In Chapter 4.2 I explain how these are realized by creating a new subexpression that contains parallel chains. This method takes three arguments—one string and two integers. The string is the expression that will be quantified, which can be either one unit or a complete expression, and the two integers are the minimum and maximum numbers of allowed repetitions. The method simply loops from minimum to maximum, with a nested loop that chains the subexpression to the appropriate lengths, and returns the complete expression.

5.4.6 void build_from_expression(...)

This is the longest and most complex method in the PMatch class, and requires some more detailed explanation. The method takes four arguments—a string, two State pointers and a vector of functors—the latter passed as reference to avoid a possibly large amount of memory consuming copies. The string is the regular expression that will be translated into a state machine. The two State pointers are pointers to the start and end states that the created state machine, or part of a state machine, will connect to. The method works in a recursive manner, according to the algorithm described in Chapter 4.2.

To explain the method, I will start by explaining the method's variables.

```
enum {ONE, ZEROORMORE, ONEORMORE, ZEROORONE, OTHER} quantifier;
int quant_min, quant_max, callback_n, current_pos = 0;
State<Functor> *new_state, *from_state = start_state;
```

```
std::string unit, subexpression;
std::vector<State<Functor>*> loose_ends;
```

Listing 5.6 – The variables of the build_from_expression(...) method.

The first variable, *quantifier*, is an *enum* with the named values *ONE*, *ZEROORMORE*, *ONEORMORE*, *ZEROORONE* and *OTHER*, which corresponds to the different quantifiers in the regular expression syntax. *OTHER* refers to the {*m,n*} quantifier.

Then there are four integer variables, called *quant_min*, *quant_max*, *callback_n* and *current_pos*. The first two holds the minimum and maximum values of the {*m,n*} when it is used, *callback_n* is used to hold the value of a functor index succeeding an @-sign, and exists simply for readability reasons, and *current_pos* is used for iterating through the regular expression string.

Two State pointers, called *new_state* and *from_state*, are used to hold the addresses of the two states corresponding to the states of the fundamental building blocks explained earlier. While *from_state* holds the address of the state it will be transitioned *from*, *new_state* holds the address of the state it will be transitioned *to*. The transition between these states can be either one single transition, or several transitions and states, depending on whether the expression is a single unit or another subexpression, and whether it is quantified.

The strings *unit* and *subexpression* are used to save the returned values from the *extract_unit(...)* and *extract_subexpression(...)* methods. They are passed either as transition conditions to the relevant State method or as the regular expression to a recursive call to itself.

Last, we have a vector of State pointers called *loose_ends*. Regular expressions often contain boolean OR-operators between units to define alternative

accepted strings, like <a><c>|<c><d>. When this happens, one has to start over at the start state at every vertical bar, and a pointer to the last state before the bar is then pushed onto the vector temporarily. When the iterator hits the end of the string, all the states with their addresses in the vector, in addition to the last one created before the end of the string, get an epsilon transition to the end state pointer if it is not *NULL*.

The rest of the method is wrapped in a while loop that continues as long as the iterator *current_pos* is less than the length of the regular expression string. In this function I use the square bracket notation to access characters in the string instead of the usual *at(...)* method, because at the end of the last iteration the code might try to access a character outside the boundaries of the string, which would throw an error with *at(...)*, but evaluates to *NULL* with square brackets—this simplifies the code in this specific case. The code inside the loop is based on a switch that evaluates the character at the position of the iterator. The switch has three cases in addition to the default case. If the character is a left angle bracket or an opening parenthesis, the code extracts the unit or subexpression and sets the value of the appropriate string, and clears the other one. The iterator is increased with the length of the unit or subexpression, so that the new iterator value will point to the character immediately following after the closing bracket or parenthesis. When this is done, a new state is created, and the *new_state* pointer is set to hold the address of the new state.

Then, a new switch evaluates the next character. There are cases responding to all the different quantifiers, including the opening curly brace as the start of the {*m,n*} quantifier, and a default case that in all other cases sets *quantifier* to *ONE*. If the character is either '*', '+' or '?', the corresponding value of *quantifier* is set, and the iterator is increased by one. If the character is '{',

however, the minimum and maximum values are extracted from the string and assigned to quant_min and quant_max respectively, and the preceding unit or subexpression is passed to the *build_quantified_subexpression* method. The return value is assigned to the *subexpression* string, the *unit* string cleared, and the iterator is increased so that it points to the character immediately following the closing curly brace.

After this switch, a series of if-clauses tests the value of *quantifier* and emptiness of the *unit* and *subexpression* strings. Based on this and the fundamental building blocks, it either uses the State methods to create the transition conditions between the states, or it calls itself recursively with the subexpression as the regular expression string, *from_state* as start state, *new_state* as end state, and of course passes the vector of functors as the last argument. After this, epsilon transitions are added, also according to the fundamental building blocks.

One of the last things that are done, is checking the next character to see if it is an @-sign. If it is, it means that the new state should be assigned a functor through its *add_functor(...)* method. Which functor is decided by trying to convert the following character, or characters, to an integer. If this succeeding character is not a number, the index will be set to zero through the functionality of the standard *atoi(...)* function. The iterator is increased to the character succeeding the @-sign. This might be an integer if the index is more than one digit, but this does not matter as it will be handled by the switch's default case. Before ending the case, *from_state* is set to the value of *new_state*.

The last case of the outermost switch, not counting the default case, is for the vertical bar character. This case simply pushes the address of the last created state onto the *loose_ends* vector, sets *from_state* to the value of the start state,

and increases the iterator by one. It is worth noting that this may seem like an error at first glance, since vertical bars are also used inside units, between sub-units, but remember that the iterator will be skipped passed units in the case above, and will never evaluate the characters inside the units, unless the regular expression is malformed.

The default case of the outermost switch only increments the iterator.

When the while loop condition no longer evaluates to *true*, meaning that the string has been iterated through to the end, all the states with their address in the *loose_ends* vector, in addition to the last state created, are assigned with epsilon transitions to the pointer given in the end state argument, if this is not *NULL*. When the first call of this recursive function finishes, the regular expression has been translated to a working nondeterministic finite-state machine.

5.4.7 void initialize(...)

This is actually three overloaded methods that differ in the definition of their functor receiving parameter. They all have the first parameter, which is the regular expression string, and the third and last parameter, which is the boolean value deciding whether the initial state will self transition regardless of input, defaulting to true, in common. The first variant takes a functor pointer, and triggers when an array of functors—and, strictly speaking, a pointer to a single functor—is passed. The second variant takes a single functor object. Both of these creates a vector of functors, where the first uses *get_highest_callback_index(...)* to decide the vector length, and the second simply makes a vector with one element, and passes it to the third overloaded method which takes a vector of functors.

The vector receiving method has only one variable, the integer *callback_n*, which is used to extract the index of any leading @-sign, and exists simply to better the readability. The first thing the method does is to clear the states vector. This is to make sure that any previously built initialized state machine is erased before building another. Second, a new state is created and pushed on the state vector, before checking if the regular expression starts with an @-sign. If it does, the index is extracted from the string, and the proper functor is assigned to the first state. This functor will be run on every restart, and if initialized with continuous it will run every time the PMatch object is stepped.

Then, *build_from_expression(...)* is called with the complete regular expression, the first state address as start state, *NULL* as the end state, and of course the vector of functors. After this, the first state is set to make a conditionless self transition through the State method *set_stay_enabled(...)* if appropriate, and the first state is marked for enabling.

The last thing that happens is that all the states are looped through to be set as enabled if they have been marked for enabling. The reason it is not sufficient to only call this method on the first state, is that the first state might have one or more epsilon transitions to other states that will be marked for enabling, and thus all should be updated.

5.4.8 void step(...)

This method exists in all of the classes except the *Sub_rule* class. It uses variadic templates, and does not do much in this class, except calling the State *step(...)* method of all the states with the unpacked arguments it received, and subsequently loop through all the states to enable those that have been marked for enabling.

5.4.9 void restart()

The *restart()* method is a very simple method that first loops through all the states to disable them, then marks the initial state for enabling, and loops through all the states to enable the ones that are marked for enabling. This restarts the PMatch object.

5.4.10 bool is_empty()

The term "empty" refers to whether the state machine contains no enabled states. If there are no enabled states, no states will ever be enabled in the future, unless the PMatch object is restarted. This can be very handy in many circumstances.

This method contains one variable—a boolean called *contains_active_states* that is initialized as *false*. Then it loops through all the states and calls the State method *is_active()*, which simply returns whether it is enabled or not. This is done in a manner where in each iteration of the loop, *contains_active_states* is set to equal itself *OR*'ed with the value of the return value of the *is_active()* method of the state in that iteration. It should be noted that this is actually a very simple optimization that takes advantage of *short circuit evaluation*, so that if *contains_active_states* is already set to *true*, the state's *is_active()* method will not be called, as C++ understands that the result of the *OR*'ing will be *true* no matter what.

The return value is the boolean inverse of *contains_active_states*, which represents whether the state machine is empty or not.

5.4.11 void debug_output_structure()

If the flag *PMATCH_DEBUG* is set through a *#define* preprocessor statement, this method is declared and can be called. It is meant for debugging purposes only, and outputs the structure of the state machine in a human readable manner. Every state is outputted with the headline "*Si*", where i is the index of the state in the vector, with its own memory address and its transition conditions formed as regular expression units together with the memory address of the state the transition points to, and whether the state has an associated functor. The method is propagated through identical and similar methods in the states, rules and sub rules, which is why the transition conditions are outputted with the memory address, and not the vector index, of the states they point to. In PMatch, the method only outputs a string of the form "Si (ADDR):\n", where i is the vector index, *ADDR*is the state's memory address and $\backslash n$ is a newline character.

The same flag is checked to see if the <*iostream*> library should be included.

5.5 The State class

This is the implementation of the actual states in the state machine. In this model, states can be enabled or disabled, and enabled states are sensitive to input passed from PMatch. When they receive an input, they start the process of going through all of their saved transition conditions to see if any of them match the input symbol. If one or more transitions do match, the state marks the target states for enabling, and disable themselves. PMatch then loops through all states to update their status, in which the states that have been marked for enabling enable themselves. This has to be done in two steps, because only the

states that are enabled should be able to enable other states, and the states somehow need to "know" whether they were just activated by a transition in this step or if they have been active all along.

The State class does not inherit *sc_module*, because that would prevent the user from being able to re-initialize the PMatch object during simulation, as the states are deleted and other states are created when the PMatch *initialize(...)* method is called.

An overview of the class members and methods follows.

```
template <typename Functor>
class State {
private:
  bool enabled, do_enable, stay_enabled, has_functor;
  std::vector<Rule<Functor>> transition_rules;
  std::vector<State<Functor>*> epsilon_transitions;
  Functor callback_function;

public:
  State();
  void add_functor(Functor _functor);
  void add_rule(std::string unit, State<Functor> *target_state);
  void add_epsilon(State<Functor> *target_state);

  template <typename... Inputs>
  void step(Inputs... inputs);

  void mark_enable();
  void update_enabled();
  void set_stay_enabled(bool _stay_enabled);
  void disable();
  bool is_active();
```

```
  #ifdef PMATCH_DEBUG
  void debug_output_structure();
  #endif
};
```

Listing 5.7 – An overview of the members and methods of the State class.

The first four members are the boolean variables *enabled*, *do_enable*, *stay_enabled* and *has_functor*. The first obviously holds whether the state is enabled, the second holds whether the state has been marked for enabling, the third is used only by the initial state as an indicator for whether it should do a self transition regardless of what the input symbol is, and the fourth holds whether the state has an assigned functor or not.

Then there is a vector of Rules. A Rule is another class in the model, and is described in further detail in its own chapter below. Every Rule contains a pointer to a state and the conditions for transitioning to that state. There is also a vector of State pointers, which are simply epsilon transitions.

The last member is the functor, specified as the data type *Functor* given from the template definition.

Again, I will go through the methods from top to bottom in the order they are written in the overview above, and explain what they do and how.

5.5.1 State()

The constructor only initializes the members to appropriate values. All the boolean values are set to false.

5.5.2 void add_functor(...)

This method takes a functor as an argument and sets the state's own functor equal to this, and sets *has_functor* to *true*.

5.5.3 void add_rule(...)

Taking a regular expression unit as the first argument and a State pointer as the second and last argument, this method creates a new Rule with these arguments as the Rule's constructor argument and pushes it onto the vector of Rules.

5.5.4 void add_epsilon(...)

This method takes a State pointer and pushes it to the epsilon transitions vector.

5.5.5 void step(...)

This is propagated down from the PMatch *step(...)* and takes the same variadic template argument. It first checks if the state itself is enabled, and if it is, it loops through the Rule vector and calls a similar *step(...)* method of the Rules. After this it disables itself, and finally it checks if *stay_enabled* is set to *true*, and if so, it marks itself for enabling—the latter can only happen if it is the initial state and the PMatch object was initialized as continuous, meaning its last argument was specifically set to *true* or omitted.

5.5.6 void mark_enable()

This method is used by another state that performs a transition to the state in question to mark it for enabling. The method sets *do_enable* to *true*, before it

loops through any epsilon transitions and calls the *mark_enable()* methods of the states pointed to.

5.5.7 void update_enabled()

After the *step(...)* method of all the states have been called, this method is called to make all the states that are marked for enabling enable themselves. It first checks if *do_enable* is set to true, and if it is, it sets *enabled* to true. If *has_functor* is *true*, the functor is called.

5.5.8 void set_stay_enabled(...)

This method is only called if it is the initial state, and only if the PMatch object was initialized as continuous. It simply sets *stay_enabled* to the boolean value passed as the argument.

5.5.9 void disable()

When restarting the state machine, all states must be disabled before the initial state is re-enabled. This method sets both *enabled* and *do_enable* to *false*.

5.5.10 bool is_active()

This method simply returns the value of *enabled*. It is used when the PMatch method *is_empty()* is called.

5.5.11 void debug_output_structure()

As for PMatch, this is only declared and defined when the flag *PMATCH_DEBUG* is defined through the *#define* preprocessor statement. It starts by calling the

identically named function of all the Rules in the Rule vector, and continues by outputting all the epsilon transitions and whether the state has an associated functor.

5.6 The Rule class

The Rule class corresponds to a regular expression unit. It is what defines the transitions and contains both a State pointer that points to the state the transition goes *to*, and the conditions that must be met for the transition to occur. A regular expression unit consists of several sub-units with boolean operators between them, and in exactly the same way does a Rule consist of several Sub_rules and boolean operators. An overview of the Rule class can be seen below.

```
template <typename Functor>
class Rule{
private:
  std::vector<Sub_rule> sub_rules;
  std::vector<char> bool_rels;
  State<Functor> *target_state;

  std::string unescape(std::string escape_string);
  std::vector<std::string> split_fields_to_vector(std::string
      field_set);
  std::vector<std::string> split_if_range(std::string field);
  std::vector<bool> recursive_compare(int input_number);

  template <typename First, typename... Rest>
  std::vector<bool> recursive_compare(int input_number, First first
      , Rest... rest);
```

```
  void create_sub_rule_from_string(std::string sub_unit);

public:
  Rule(std::string regexp_unit, State<Functor> *_target_state);

  template <typename... Inputs>
  void step(Inputs... inputs);

  #ifdef PMATCH_DEBUG
  void debug_output_structure();
  #endif
};
```

Listing 5.8 – An overview of the members and methods of the Rule class.

As usual, I will start by explaining the members. The first member is a vector of Sub_rules. The Sub_rules contain the conditions for a transition to occur, and have methods for matching against input. The second member is a vector of characters. These characters equal the boolean operators that separate the sub-units in a unit, where increasing index correspond to their order when reading from left to right. This means that the boolean relationship between the Sub_rules in the Sub_rule vector with index n and $n+1$ is saved in the character vector at index n.

The methods of the class, from top to bottom in the overview, are explained below.

5.6.1 std::string unescape(...)

This method takes a string and deletes all the escaping backslashes, and returns the result. This is done by looping through the string until a backslash is found, and then deleting the backslash and skipping the succeeding character. Skipping the succeeding character avoids deleting escaped backslashes.

5.6.2 std::vector<std::string> split_fields_to_vector(...)

The field sets need to be split up into separate fields, and this method does exactly that. It takes the field set without enclosing square brackets, and without a leading caret, and splits the set of comma separated fields into a vector of field strings. It ignores escaped commas. The resulting vector is returned.

5.6.3 std::vector<std::string> split_if_range(...)

This method takes a field string, and if it is a range field, it splits it into a two element vector containing the upper and lower bounds of the range. If it is not a range, it returns a vector of length zero.

5.6.4 std::vector<bool> recursive_compare(...)

The *recursive_compare(...)* method actually consists of two overloaded methods, where one has an empty body and takes only an integer as argument, while the other takes an integer, a template argument and a variadic template argument. It works, as the name suggests, in a recursive manner, comparing the non-integer arguments to the transition conditions of the Rule. The single template argument is compared to the Sub_rule corresponding to the index

passed as the integer argument, and then it calls itself with the integer argument incremented and the unpacked argument pack as the other arguments. When the method calls itself with a zero length argument pack, it effectively calls the empty method which ends the recursion. It ends up returning a vector of boolean values representing the results of comparing each input—index zero is the result of comparing the first input to the first Sub_rule, index one is the result of comparing the second input to the second Sub_rule, and so on.

The method has two variables, which are both boolean vectors. One is called *result* and another *append*. First it matches the single template argument with the appropriate Sub_rule, and pushes the result on the *result* vector. Second, it calls itself recursively like explained above and saves the result in the *append* vector. The third step is appending *append* to *result*, and finally it returns *result*.

5.6.5 void create_sub_rule_from_string(...)

This method takes a regular expression sub-unit passed as a string and creates a Sub_rule. The method has five variables; four string vectors and one boolean. The first two string vectors are *values* and *ranges*. These are used to contain the single values and the upper and lower bounds of the ranges that may be defined in the sub-unit. The other string vectors, *split_fields* and *new_range*, are used to temporary hold fields and ranges—this prevents calling a few functions several times. The last variable is a boolean variable called *invert*, which is initialized to *false* and is set to *true* if the sub-unit is a field set starting with a caret—this would match any input *except* the values and ranges defined by the fields.

The method starts by checking if the sub-unit is nothing more than a dot character. If it is, then a wildcard transition is made by creating a new Sub_rule

without constructor arguments and pushing it onto the *sub_rule* vector and class member. If it is not a dot character, it checks if the first character is an opening square bracket. If so, this means that the sub-unit is a field set, and the square brackets are trimmed. The next step is to check if the following character is a caret, and if it is, *invert* is set to *true* and the caret is trimmed.

The field set is then passed as an argument to the *split_fields_to_vector(...)* method, which returns a string vector of fields. This vector is saved to the local variable *split_fields*. Next, the method loops through all the fields in *split_fields*, passing them one at the time to *split_if_range(...)* and temporary saving the returned vector in *new_range*. If *new_range* has no elements, it means that the field was not a range field, and the field has its escaped characters unescaped before it is pushed onto the *values* vector. If *new_range* does have elements, it means that the field is a range field, and the lower and upper bounds of the range have their escaped characters unescaped before being pushed onto the *ranges* vector in that order. After the loop is finished, a Sub_rule is created with these values and ranges and pushed onto the *sub_rules* vector.

If the sub-rule is not in its entirety a dot character, and the first character of the sub-unit is not an opening square bracket, then the sub-unit must consist of a single value. This value has its escaped characters unescaped before it is pushed onto the *values* vector, and a new Sub_rule is created based on this value and pushed onto the *sub_rule* vector.

5.6.6 Rule(...)

The constructor takes a regular expression unit as the first argument, and a State pointer as the second argument. First, it sets the Rule's target state equal to the incoming state pointer, and then it trims the angle brackets off

the regular expression unit. The stripped unit is then iterated through, and when it reaches a boolean OR- or AND-operator, in the form of the not-escaped characters '|' and '&' respectively, it passes the preceding field set or value to *create_sub_rule_from_string(...)*. Then it pushes the boolean operator character onto the member vector containing all the boolean relationships between sub-units, and saves the character position after the operator in order to extract the next field. When the loop ends, the last field is passed to *create_sub_rule_from_string(...)*.

5.6.7 void step(...)

This is where the propagation of the *step(...)* function from PMatch via State ends. This *step(...)* method uses a boolean variable called *condition_met* to keep track of whether the transition conditions are met, and two boolean vectors to save the results of matching the Sub_rules and the results of *AND*'ing the appropriate results, called *sub_results* and *anded* respectively.

First, *condition_met* is initialized to false, and *sub_results* is initialized to the return value of *recursive_compare(...)* called with zero as the first index together with the unpacked input arguments. After *sub_results* has been set, the *AND*'ing and *OR*'ing between the results needs to be done. As I have already mentioned, *AND* has higher precedence than *OR*, so all the *AND*'ing is done first. This is done by first appending the first result in *sub_results* to the *anded* vector, and then looping through the character vector containing the boolean operators. When the operator at the current index is the AND-operator, the value in *that index plus one* in *sub_results* is *AND*'ed with the last value of the *anded* vector, and the result overwrites the last element of *anded*. When the operator at the current index is the OR-operator, the value in *that index plus*

one in *sub_results* is appended at the end of *anded*.

When this process is over, all the elements in *anded* are *OR*'ed together, and the result is written to *condition_met*. If *condition_met* ends up *true*, the Rule's target state is marked for enabling.

5.6.8 void debug_output_structure()

Again, this is only declared and defined if the flag *PMATCH_DEBUG* is defined with the *#define* preprocessor statement. The method simply outputs its own target state, and calls the similar method in its Sub_rules.

5.7 The Sub_rule class

Sub_rules in the state machine correspond to sub-units in the regular expressions. They contain whether they are a wild card, whether they are inverted or not, the different values and the different ranges. But they also contain methods for matching input to these values and ranges, as well as methods for building the Sub_rule from a sub-unit. An overview of the code follows.

```
class Sub_rule {
private:
  bool invert;
  bool wildcard;
  std::vector<std::string> single_values;
  std::vector<std::string> ranges;

public:
  Sub_rule(bool _invert, std::vector<std::string> _single_values,
      std::vector<std::string> _ranges);
  Sub_rule();
```

```
  template <typename Input>
  bool match(Input input);

  #ifdef PMATCH_DEBUG
  std::string debug_escape_characters(std::string chars);
  void debug_output_structure();
  #endif
};
```

Listing 5.9 – An overview of the members and methods of the Sub_rule class.

There are four members—the boolean variables *invert* and *wildcard*, and the string vectors *single_values* and *ranges*. What information they hold should be fairly obvious, but I can mention that *ranges* always contain an even amount of elements. A range is saved in *ranges* as two separate elements, the first being the lower bound and the succeeding being the upper bound. The methods of the class are rather simple, and do not need a very detailed walk through. Nonetheless, the methods are described below.

5.7.1 Sub_rule(...)

The constructor is overloaded, where one takes no arguments and the other takes a boolean value that represents whether the sub-rule is inverted, and two string vectors containing the values and ranges of input. In the former case the Sub_rule becomes a wildcard, and thus *wildcard* is set to *true* and *invert* is set to *false*. In the latter case, *wildcard* is set to false, and *invert* is set to the corresponding argument, as are the *single_values* and *ranges* vectors. Note that the values that will be compared to the input symbols are saved as string

representations. Casting to the input data type of the *step(...)* functions are done when comparing the input to the transition conditions.

5.7.2 bool match(...)

This method takes a templated argument. It can take absolutely any data type, but the user must make sure that casting process from the string representation to the input data type is correct—this is mentioned elsewhere, but a walk through of the casting functions can be found in chapter 5.8 has only one variable, the boolean *matched*. This is initially set to the same value as *wildcard*, and the reason for this will become clear. After initialization, the first thing that happens is a looping through the *single_values* vector, where the input is compared to the values in the vector, after being casted to the input data type. The *OR*'ing of the boolean variable *matched* and the result of the comparing is written back to *matched*. Again, this is done in a way that short circuits evaluation if *matched* is already set to *true*, either by a successful match or the Sub_rule being a wildcard.

Next, the same kind of matching is done with the elements of *ranges*, where the evaluation of the input being greater than or equal to the range's lower bound is *AND*'ed with the evaluation of the input being less than or equal to the upper bounds, and this result is *OR*'ed with the accumulated result in *matched*, and written back to *matched* itself. Again this takes the advantage of evaluation short circuiting.

Note that if one is using self defined data types, one must both be sure that the casting happens the way it should, and also that the data type supports equality testing, as well as the *greater than or equal to* and *less than or equal to* operations if ranges are used. In some cases this might demand that the user

overloads these operations in the data type definition.

5.7.3 std::string debug_escape_characters(...)

Also only being declared and defined if the *PMATCH_DEBUG* flag is set with the *#define* preprocessor statement, this method loops through the argument string and escapes reserved characters. This is used to prevent any ambiguity when outputting the conditions of the transition.

5.7.4 void debug_output_structure()

As above, this is only declared and defined if the *PMATCH_DEBUG* flag is set. It does nothing more than output the transition conditions, using the method described immediately above to escape reserved characters.

5.8 The casting functions

The casting is a very important part of the state machine functionality. If the values of the regular expressions were to be saved as the data type they would later be compared with, the structure of the system would become much more complicated—for instance, it would prevent Rule's from having vectors of Sub_rules, as Sub_rules for different inputs would be structurally different in memory. It would also become excessively verbose on the user end when using many inputs, and would prevent the user from reinitializing a PMatch object for using inputs of different data types.

For the casting to work properly with any data type, I have made a main, general, templated casting function which uses a *stringstream*, and several ex-

plicit template specifications for certain otherwise problematic data types. The main function works by first inserting the string passed as argument into a stringstream and then extract it with the >> operator into a templated variable which it returns. The problem, however, occurs when the template argument is a string, a c-type string (character pointer with null termination) or a character. The extraction operator of these data types is programmed to stop at whitespace characters. For strings and c-type strings, that means that it would only return the first "word" of the string, ignoring leading whitespaces. For the char data type, it would simply not convert whitespace characters.

Because of this, explicit template specializations were made for *std::string*, *char* of types signed and unsigned, both const and not const, and the same with *char pointers*. Since the string parameter of the function is specified to pass by reference, the char pointer specifications simply return the pointer to the string's own c-string representation, and the char specifications return the character at index zero in the string. These explicit template specializations allow for using whitespace characters and strings with any kind of whitespace as input. I refer the reader to the appendix for the code, as an overview here would be unnecessary and superfluous.

To prevent any interference with other libraries or user code, these functions are enclosed in their own namespace. They were made as global functions first and foremost because C++ does not allow partial template specifications inside classes, and only allows them at namescape scope[13]. Having them global also makes them easily overloaded with user defined functions, and the functions could be used from outside the Sub_rule class, if necessary or desirable.

When using user defined data types as input, it is absolutely necessary to either overload the stream extraction operator of that data type, or define an

explicit template specialization for that data type in the *pcast* namespace.

Chapter 6

Examples of use

6.1 Inheriting PMatch, two boolean inputs

In my last related project, I used the example of performing one task if the input was four consecutive ones, and something else if the input was only three consecutive ones. I decided to use a conceptually, architecturally and functionally similar example here to show the difference between the two systems. The difference in this context is that one can have several inputs, that the main system—the state machine then and PMatch now—does not use a SystemC interface unless one is defined by the user, and that it is much simpler and less tedious to initialize.

The example consists of one top module that contains a generator module, a datapath module and the pattern matching module—and the system is clocked and synchronous. The architecture can be seen in Figure 6.1. The generator produces output consisting of two boolean values as input to the pattern

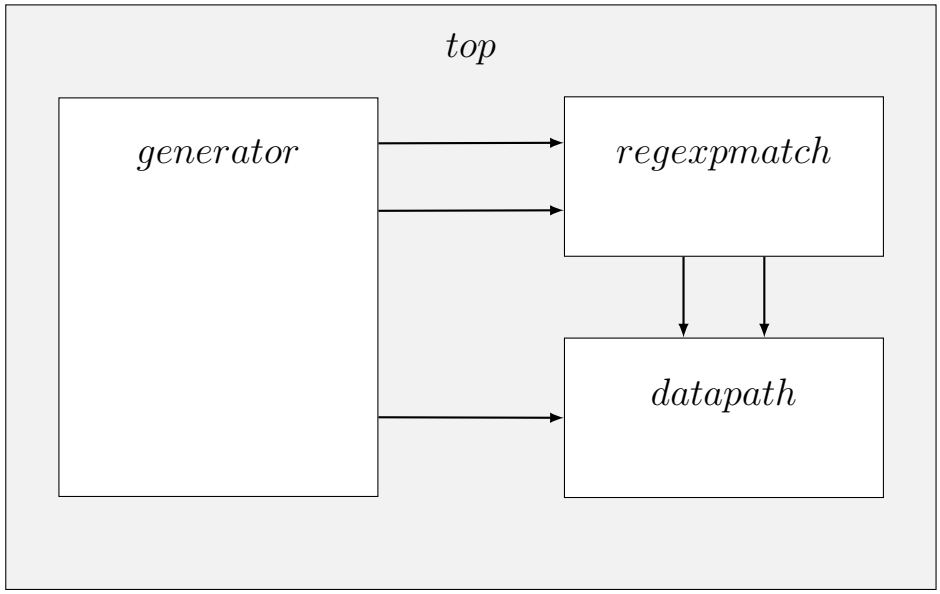

Figure 6.1 – The architecture used in the four double ones versus three double ones and a one-zero-combination example. The clock signal is not shown.

matching module, and one 16 bit SystemC integer (sc_int<16>) value to the datapath module, on every negative edge of the clock signal. The datapath module performs a clocked shifting of the incoming integer into a local register with a length of four elements on positive clock edges, and is sensitive to two input signals that triggers two different mathematical operations on the integers in the register. In this example, the datapath module simply outputs the results to the screen as a confirmation, and does nothing more with the results.

The pattern matching class inherits PMatch, and is called *Regexpmatch*. It initializes itself with the proper regular expression and lambda functions,

and omits the *continuous* argument, meaning it defaults to *true*. On every positive edge on the clock signal, it steps itself with the incoming inputs. If the input is four consecutive double ones, it sets one of the two signals used for performing calculations in the datapaths high, and resets itself—the resetting is to prevent every following one to trigger the same function. Also remember that the surrounding scope of the lambda function remains what it was when it was defined, which is why we can conveniently use the *this* pointer. Note that the *this* pointer does not refer to the lambda function object. Calling *restart()* from within the lambda without specifying an object does not work.

If the input is three consecutive double ones followed by the first input being one and the other zero, the other lambda function is triggered—resetting is not necessary in this case, but it could of course be done for uniformity and readability reasons. This lambda function sets the other calculation triggering signal high. Both of these calculation signals are brought low on the next negative clock edge.

It is worth noting that the method of initializing vectors used in this example—here done directly in the *initialize(...)* argument list—is a C++11 feature called *initializer lists*. This feature allows for many container structures and other classes to be initialized with the same syntax one is familiar with being used for initializing arrays and structs.

The code listing for this example, the console output, and the waveform showing the signal changes throughout the simulation follows.

```
#include <systemc.h>
#include "regexpmatch.h"
#include "generator.h"
#include "datapath.h"
```

```
using namespace std;

class Top : public sc_module{
public:
  Regexpmatch match;
  Generator generator;
  Datapath datapath;
  sc_signal<bool> regex_input_1, regex_input_2, calculation_1,
      calculation_2;
  sc_signal<sc_int<16>> integer_data;
  sc_clock clk;    // 10 ns period clock

  Top(sc_module_name name) : sc_module(name), match("match"),
      generator("generator"), datapath("datapath"), clk("clk", 10,
      SC_NS){

    generator.out_1(regex_input_1);
    generator.out_2(regex_input_2);
    generator.out_data(integer_data);
    generator.clk(integer_data);

    match.in_1(regex_input_1);
    match.in_2(regex_input_2);
    match.out_calc_1(calculation_1);
    match.out_calc_2(calculation_2);
    match.clk(clk);

    datapath.in_calc_1(calculation_1);
    datapath.in_calc_2(com_4);
    datapath.in_data(calculation_2);
    datapath.clk(clk);
  }
```

```
};


int sc_main(int argc, char* argv[]){
  Top top("top");

  sc_trace_file *fp;                    // VCD filepointer
    fp=sc_create_vcd_trace_file("wave");// Create wave.vcd file
    sc_trace(fp, top.clk, "clk");              // Add signals to
        trace file
    sc_trace(fp, top.regex_input_1, "regex_input_1");
    sc_trace(fp, top.regex_input_2, "regex_input_2");
    sc_trace(fp, top.calculation_1, "calculation_1");
    sc_trace(fp, top.calculation_2, "calculation_2");
    sc_trace(fp, top.integer_data, "integer_data");

  sc_start(130, SC_NS);

    sc_close_vcd_trace_file(fp);        // close wave.vcd

  return 0;
}
```

Listing 6.1 – SystemC code for the top module.

```
#include "../pmatch/pmatch.h"
#include <iostream>
#include <systemc.h>
#include <functional>

using namespace std;

class Regexpmatch : public PMatch<>{
```

```
private:

public:
  sc_in<bool> clk, in_1, in_2;
  sc_out<bool> out_calc_1, out_calc_2;

  Regexpmatch(sc_module_name name) : PMatch<>(name){
    SC_HAS_PROCESS(Regexpmatch);
    SC_METHOD(step_it);
    sensitive << clk;
    dont_initialize();

    initialize("<1&1><1&1><1&1>(<1&1>@0|<1&0>@1)", { [&](){
        out_calc_1.write(1); this->restart();}, [&](){out_calc_2.
        write(1); this->restart();} });
  }

  void step_it(){
    if(clk.posedge()){
      step(in_1.read(), in_2.read());
    }
    else{
      out_calc_1.write(0);
      out_calc_2.write(0);
    }
  }
};
```

Listing 6.2 – SystemC code for Regexpmatch, which inherits PMatch.

```
#include <systemc.h>

class Generator : public sc_module{
```

```
public:
  sc_in<bool> clk;
  sc_out<bool> out_1, out_2;
  sc_out<sc_int<16>> out_data;

  sc_bv<10> bits_1, bits_2;
  sc_int<16> data[10];
  int counter;

  Generator(sc_module_name name) : sc_module(name){
    SC_HAS_PROCESS(Generator);
    SC_METHOD(main);
    sensitive << clk.neg();
    dont_initialize();

    bits_1 = "1111101111";
    bits_2 = "1111010111";
    for(int i = 0; i < 10; i++){
      data[i] = i+1;
    }
    counter = 0;
  }

  void main(){
    if(counter < 10){
      out_1.write(bits_1.get_bit(counter));
      out_2.write(bits_2.get_bit(counter));
      out_data.write(data[counter]);
      ++counter;
    }
  }
};
```

Listing 6.3 – SystemC code for the Generator module.

```
#include <iostream>
#include <systemc.h>

class Datapath : public sc_module{
public:
  sc_in<bool> clk, in_calc_1, in_calc_2;
  sc_in<sc_int<16>> in_data;
  deque<sc_int<16>> reg;

  Datapath(sc_module_name name) : sc_module(name), reg(4,0){
    SC_HAS_PROCESS(Datapath);
    SC_METHOD(push_reg);
    sensitive << clk << in_calc_1 << in_calc_2;
    dont_initialize();
  }

  void push_reg(){
    if(clk.posedge()){
      reg.pop_back();
      reg.push_front(in_data.read());
    }
    if(clk.read() && in_calc_1.posedge()){
      cout << reg.at(2) * reg.at(1) + reg.at(0) << endl;
    }
    if(clk.read() && in_calc_2.posedge()){
      cout << (reg.at(3) + reg.at(2)) * (reg.at(1) + reg.at(0)) <<
          endl;
    }
  }
```

```
};
```

Listing 6.4 – SystemC code for the Datapath module.

```
                SystemC 2.2.0 --- Mar  3 2011 12:17:11
        Copyright (c) 1996-2006 by all Contributors
                        ALL RIGHTS RESERVED
WARNING: Default time step is used for VCD tracing.
21
82
```

Listing 6.5 – The simulation results from the four double ones versus three double ones and a one-zero-combination example.

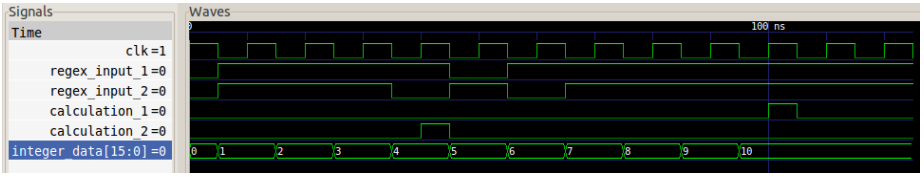

Figure 6.2 – The waveform from the simulation of the four double ones versus three double ones and a one-zero-combination example.

6.2 Using PMatch as a module, a HTTP server

Sometimes it can be just as convenient to use PMatch as a stand alone module, as to define a class that inherits it. To illustrate how this is done and to give some other ideas about what PMatch can be used for, this example defines a simple server that accepts and reacts to HTTP requests. Sockets and details about the communication protocols is not included, but a client and server with proper interfaces are defined and implemented.

The server accepts GET and HEAD requests. These request types and their corresponding responses are largely similar, but where a GET request results in the contents of a file being returned in the response, the HEAD request results in only the response headers.

An ordinary but simple GET request may look like this:

```
GET /path/index.html HTTP/1.1
Host: example.com
Referer: http://example.com/index.html
User-Agent: SomeAgent/1.0
```

Listing 6.6 – An ordinary but simple HTTP GET request.

First, we have the keyword GET, where we in this case also accept HEAD. This is followed by the URL relative to the domain root, and the characters '*HTTP/*' followed by the HTTP version. The version is either 1.1 or 1.0. The next lines follow the pattern of '*keyword: value*', and there are a lot of possible keywords that are of no interest in this example that may come in any which order. The request ends with an empty line. What we need to extract from the request is the GET or HEAD keyword, the URL, the HTTP version and the host. Any other sets of keywords and values needs to be ignored.

Since a server often hosts the files of different domains, it is not unusual to have one folder for each domain with its name identical to the domain name. The files that belong to this domain are all in this folder, where the URL represents the relative path. The HTTP version is used in the response.

The headers of a normal response to a GET request—or, identically, the entire response of a HEAD request—may look like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

```
Content-Length: 179
```

Listing 6.7 – The headers of a normal response to a GET request.

Following the HTTP version is the status code. When the file is found, this is usually '*200 OK*'. If the file is *not* found, the status code is '*404 Not Found*', and the data is usually a standard HTML page explaining that the file was not found. The next line describes the mime type of the response data, and the last one describes how many bytes of data the response is, excluding the headers. Many more sets of keywords and values can be included, but this suffices for this example.

The server is realized through initializing PMatch with a somewhat elaborate regular expression, a set of lambda functions operating on an input buffer to store values, and a method for initiating a request that restarts the PMatch. The regular expression is case sensitive, but it could easily be made case insensitive by exchanging all values of a single character with a field set consisting of both upper and lower case. The request is made character by character, and I will break down the regular expression into parts and walk through it. Here, $\backslash n$ is used instead of an actual newline character for readability reasons.

`(<G><E><T>@1|<H><E><A><D>@2)` : This accepts GET and HEAD, and each triggers their own lambda function. This function simply saves the request type.

`< >+@0<[^ ]>+< >@3` : First it accepts one or more spaces, with each space triggering a lambda function that clears the buffer. Then it accepts one or more characters that are *not* spaces, followed by another space. This is the URL. On the space succeeding the URL, a lambda function is triggered that saves the buffer except for the last space into a URL holding variable.

`< >*<H><T><T><P></><1><\\.>(<0>@4|<1>@5)< >*<\n>` : This part starts

by accepting zero or more spaces, followed by the string '*HTTP/1.*'. Then it accepts either '*0*' or '*1*', where each trigger their own lambda function saving the character in a variable as the HTTP version. This is followed by zero or more spaces and a newline character. This concludes the first line of the request.

`(<H><o><s><t><:>< >*@0<[^\n]>+<\n>@6|<[^\n]>+<\n>)*` : This subexpression has two alternatives. The first is the *Host* keyword followed by a colon, and zero or more spaces. After the colon, the lambda function with index zero is triggered, and will be triggered for every space received. This lambda function clears the buffer. The next unit matches one or more characters that are *not* newlines, and the following newline triggers a lambda function that saves the host value in a variable. The other alternative matches any series of characters followed by a newline, and is needed to accept but still ignore any other keyword and value pairs. The subexpression can accept such patterns zero or more times. Also note that it would not in fact be very complicated change the expression to accept any keyword and value pair, and save these in a standard std::map<std::string, std::string> variable.

`<\n>@7` : The last part of the regular expression reacts at the first blank line and executes the response generation.

The response generating method uses the saved values to try and open the file, and responds with the file contents and proper headers if it exists, and responds with a standard 404 page with the proper headers if it does not. If the request is a HEAD request, only the headers are sent in return.

The example defines two modules—the server and a client. It is a very simplistic model without complex communication protocols and such, and the regular expression could easily be made more robust with regards to whitespace, as well as accepting more keyword and value pairs, accepting URL encoded GET

arguments and so on. But the point of this example is not to make a complete and working HTTP server, but to show a simple example of how PMatch can be used to define fairly complex models using nondeterminism without a lot of code. The code of this example is contained in a single file, and the complete code is listed below. An example of a response after sending a successful GET request is also shown.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <functional>
#include <systemc.h>
#include "../pmatch/pmatch.h"

using namespace std;

class Httpserver_if : public sc_interface{
public:
  virtual void start_request() = 0;
  virtual void request_write(char) = 0;
};

class Client_if : public sc_interface{
public:
  virtual void response_write(char) = 0;
};

class Httpserver : public Httpserver_if, public sc_module{
public:
  PMatch<> pmatch;
  sc_port<Client_if> client_p;
  string buffer, address, host;
```

```
char http_ver_1;
enum {GET, HEAD} req_type;

void send_response(){
  string file_contents;
  char file_size[20];
  string response = "HTTP/1.";
  response += http_ver_1;
  response += " ";
  ifstream file;

  file.open(host + address);

  if(file.is_open()){
    getline(file, file_contents, '\0');
    sprintf(file_size, "%d", file_contents.length());
    response += "200 OK\n";
    response += "Content-Type: text/html\n";
    response += "Content-Length: ";
    response += file_size;
    response += "\n\n";
    if(req_type == GET){
      response += file_contents;
    }
  }
  else{
    sprintf(file_size, "%d", 197 + address.length());
    response += "404 Not Found\n";
    response += "Content-Type: text/html\n";
    response += "Content-Length: ";
    response += file_size;
    response += "\n\n";
    if(req_type == GET){
```

```
        response += "<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML
            2.0//EN\">\n";
        response += "<html><head>\n";
        response += "<title>404 Not Found</title>\n";
        response += "</head><body>\n";
        response += "<h1>Not Found</h1>\n";
        response += "<p>The requested URL" + address + " was not
            found on this server.</p>\n";
        response += "</body></html>\n";
      }
    }

    for(int i = 0; i < response.length(); ++i){
      client_p->response_write(response.at(i));
      wait(10, SC_NS);
    }
  }

  Httpserver(sc_module_name name) : sc_module(name), pmatch("pmatch
      "){
    vector<function<void()>> lambdas = {  [&](){buffer.clear();},
                          [&](){req_type = GET;},
                          [&](){req_type = HEAD;},
                          [&](){address = buffer; address.erase(
                              address.length() - 1);},
                          [&](){http_ver_1 = '0';},
                          [&](){http_ver_1 = '1';},
                          [&](){host = buffer; host.erase(host.length
                              () - 1);},
                          [&](){send_response();} };

    pmatch.initialize("(<G><E><T>@1|<H><E><A><D>@2)< >+@0<[^ ]>+< >
        @3< >*<H><T><T><P></><1><\\.>(<0>@4|<1>@5)< >*<\n>(<H><o><s
```

```
        ><t><:><␣>*@0<[^\n]>+<\n>@6|<[^\n]>+<\n>)*<\n>@7", lambdas,
          false);
  }

  void start_request(){
    buffer = "";
    pmatch.restart();
  }

  void request_write(char ch){
    buffer += ch;
    pmatch.step(ch);
  }
};

class Client : public Client_if, public sc_module{
public:
  sc_port<Httpserver_if> http_p;

  Client(sc_module_name name) : sc_module(name){
    SC_HAS_PROCESS(Client);
    SC_THREAD(do_request);
  }

  void do_request(){
    const char* request = "GET␣/index.html␣HTTP/1.1\nHost:␣example.
        com\nReferer:␣http://127.0.0.1/\n\n";

    http_p->start_request();
    while(*request){
      http_p->request_write(*request);
      ++request;
      wait(10, SC_NS);
```

```
    }
  }

  void response_write(char ch){
    cout << ch;
  }
};


int sc_main(int argc, char* argv[]){
  Httpserver server("server");
  Client client("client");

  client.http_p(server);
  server.client_p(client);

  sc_start();

  return 0;
}
```

Listing 6.8 – SystemC code for a client and HTTP server – the latter using a PMatch object.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 179

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>Example webpage</title>
</head><body>
<h1>An example</h1>
```

```
<p>This is an example webpage.</p>
</body></html>
```

Listing 6.9 – The HTTP response from a successful GET request in the HTTP server example.

6.3 User defined functors

Both of the above examples use lambda functions, but as I have pointed out, one can just as well use other user defined functors. This example is a minimalistic example that shows one way this can be done.

Here, a class called *A_functor* is defined with a constructor that takes an integer used to set the functionality of the functor, and an integer pointer which is simply used to manipulate an integer outside of its own scope. The functor contains only these two simple members, so no overloading of the assignment operator is needed. The code is very simple and can be seen below, followed by the output of the simulation.

```
#include <iostream>
#include <vector>
#include <systemc.h>
#include "../pmatch/pmatch.h"

using namespace std;

class A_functor{
public:
  int functionality;
  int *pointer;
```

```
  A_functor(){};

  A_functor(int functionality_, int *pointer_){
    functionality = functionality_;
    pointer = pointer_;
  }

  void operator()(){
    if(functionality == 0){
      *pointer = *pointer + 3;
    }
    else if(functionality == 1){
      *pointer = *pointer - 5;
    }
  }
};

class Some_class : public sc_module{
public:
  PMatch<A_functor> pmatch;
  int some_int, some_other_int;

  Some_class(sc_module_name name) : sc_module(name), pmatch("pmatch
      "){
    SC_HAS_PROCESS(Some_class);
    SC_THREAD(run);

    some_int = 10;
    some_other_int = 20;
    vector<A_functor> functors = {A_functor(0, &some_int),
        A_functor(1, &some_other_int)};

    pmatch.initialize("<a><b>@0<c>@1", functors);
```

```
  }

  void run(){
    pmatch.step('a');
    pmatch.step('b');
    pmatch.step('c');

    cout << some_int << " " << some_other_int << endl;
  }
};


int sc_main(int argc, char* argv[]){
  Some_class object("object");

  sc_start();

  return 0;
}
```

Listing 6.10 – SystemC code for an example using user defined functors.

```
                SystemC 2.2.0 --- Mar  3 2011 12:17:11
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED
13 15
```

Listing 6.11 – The output generated by the simulation of the example using user defined functors.

Chapter 7

Discussion

Although the PMatch library works as specified and expected, some improvements could, and perhaps should, be made. In this chapter I will discuss several different points that may improve this system in some way or another. Some regard optimization and some regard increased functionality.

7.1 Optimizing the + quantifier

The fundamental building block of the *one or more* quantifier is currently realized as a transition condition based on the quantified unit from the old state to the new state, and a self transition on the new state with the same unit. In the case where the old state has a functor associated with it, this is the way it still needs to be. But if the old state does *not* have an associated functor, this building block can in principle be replaced with a different one that can potentially save the complete state machine from a lot of unnecessary states if the

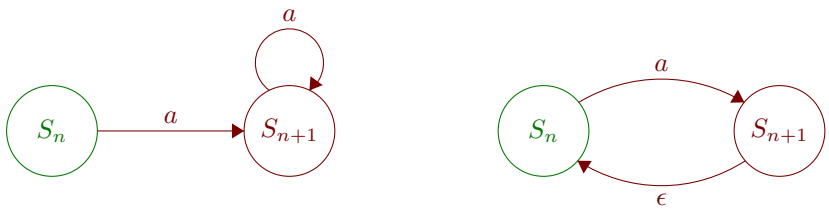

Figure 7.1 – The two alternatives of the fundamental building block for the *one or more* quantifier. The left shows the current implementation, which will have to remain as long as there is a functor associated with state S_n; the right shows the potentially state saving alternative.

quantifier quantifies a subexpression rather than a single unit. The alternative building block would consist of the same transition from the old state to the new state, but instead of the self transition on the new state, there would be an epsilon transition back the old state. This will obviously not work properly if the old state has a functor.

Both of these building blocks can be seen in Figure 7.1.

To use an extreme example, the regular expression

<a>(<b><c><d><e><f><g><h><i><j><k><l><m>)+<n>@0

would end up in its state machine form with almost half the amount of states if the proposed alternative building block was used, compared with how it would end up with the current solution. I therefore think it would be a good idea to implement this in the future—especially if area consumption is to be considered in analyses at this stage.

7.2 Bypassing epsilon transitions

Another, perhaps more complicated, method of optimization is epsilon bypassing. This would not have such a dramatic effect, but could still remove a few unnecessary states.

When units or subexpressions are *OR*'ed, which is also the case when using the $\{m,n\}$ quantifier, the current method is to keep building the state machine until an OR-operator is encountered, leave the last state temporarily hanging, and backtrack to the initial state of that recursive run and start over at the first state with the rest of the expression. When the end of the subexpression is encountered, all the states left hanging, including the last state made before the end of the expression, are connected to the supplied end state with epsilon transitions. This makes it possible to associate functors to the last state of such an *OR*'ed "string" of units, but if a functor is not associated with the last state, the last transition of that state may just as well go directly to the end state instead. The difference is shown in Figure 7.2.

This optimization could be somewhat efficient when having regular expressions with a lot of OR-operators, or if $\{m,n\}$ quantifiers where n is much larger than m is often used.

Epsilon bypassing could also be done in the case of the *zero or more* quantifier, as long as neither the old state nor the new state have associated functors. This is done by simply dropping the new state and using the unit as the basis for a self transition on the old state. The difference is shown in Figure 7.3.

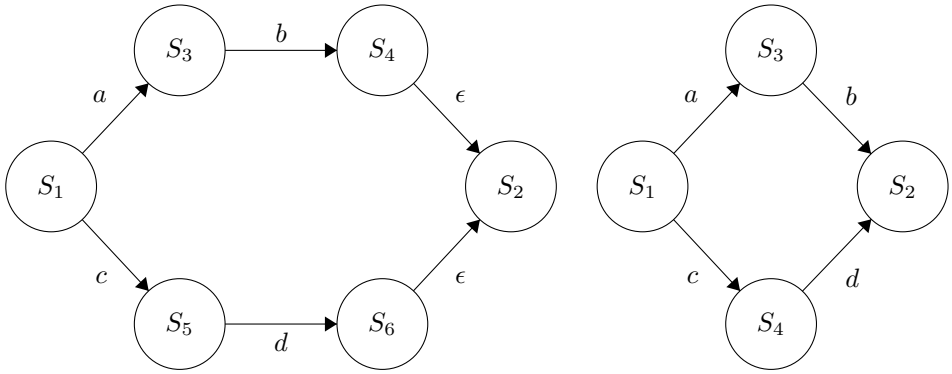

Figure 7.2 – The difference between no epsilon bypassing and with epsilon bypassing on a "two-threaded" expression. Without epsilon bypassing is shown on the left, and with epsilon bypassing is shown on the right. This operation assumes no functor association with states S_4 and S_6 on the left state machine.

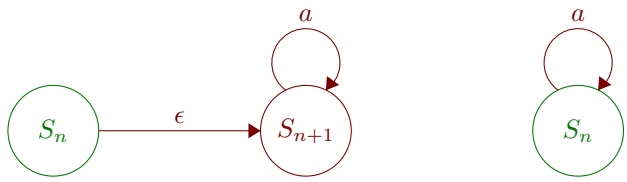

Figure 7.3 – The difference between no epsilon bypassing and with epsilon bypassing on a *zero or more* fundamental building block. Without epsilon bypassing is shown on the left, and with epsilon bypassing is shown on the right. This operation assumes no functor association with either of the states.

7.3 Introducing escape sequences

It could be practical to introduce a few escape sequences. An escape sequence is written as a backslash preceding a character, where that combination is interpreted as something in particular. If one is to write a regular expression unit that contains, say, the newline or tabulator character, one has to use the actual newline or tabulator character. This does not make much of a difference when entering the string directly in the C++ code, where these characters have their own escape sequences in the C++ language—\n and \t—but it can turn out to be very unpractical when reading a regular expression from a file, or the console input, for instance. Having defined escape sequences for whitespace characters in particular would most likely be a good idea, and while one is at it, it might be practical to introduce other typical escape sequences from regular expression syntaxes as well—such as \d for digits, \w for alphanumeric characters and underscore, and \s for any whitespace character.

Having an escape sequence for newline could be especially useful to prevent problems regarding to the two conflicting newline standards—the carriage return and linefeed characters associated with Windows systems, and the single linefeed character associated with Unix-like systems.

7.4 Two more quantifiers

In Perl syntax, and many syntaxes derived from the Perl syntax, there are two more variations of the specific quantifier currently implemented as $\{m,n\}$ alone. These are $\{n,\}$, which means *at least* n repetitions, and $\{n\}$, which means *exactly* n repetitions. I believe adopting this syntax could be very useful. Both can be

accomplished with the existing syntax—just like {*m,n*} can be accomplished by *OR*'ing several subexpressions—but it could make the regular expressions less tedious to write and easier to read.

7.5 Analyzing expressions, and clearer error messages

PMatch does not check for malformed regular expressions, and as a result, a small error in the regular expression string might lead to highly unexpected behavior instead of a clear error message. Sometimes the state machine will run, but not at all like expected; sometimes everything will seem to work until PMatch is stepped, and the user is supplied with an vector out-of-range error or something similar; and sometimes a similar non-descriptive error for the user might come at compile time.

Implementing a method for analyzing the regular expression for syntactical errors, as well as introducing try-catch blocks in the code at large to output errors that are more easily understood by the end user—and the library developer, for that matter—could prove beneficial.

7.6 Two more range definitions

With the current regular expression syntax, there is no way of declaring that the input should be greater than or equal to some value without also specifying that it should be less than or equal to some other value—as well as the opposite. This could prove to be useful, so I suggest that it should be possible to write half-defined ranges as '*n*-' and '-*n*', where *n* is an arbitrary value, which represent

greater than or equal to n and *less than or equal to n* respectively.

To implement this, one would most likely have to introduce two more vectors in the Sub_rule class as well as giving its constructor two more parameters, and edit a few methods in the Rule class.

Chapter 8

Conclusion

A SystemC library for creating nondeterministic finite-state machines from regular expressions was created, and a specific regular expression syntax has been developed to meet the needs of this system. The library works as specified, and some additional features have been added during development. The PMatch library can use any data type as input, and supports an indefinite amount of inputs. Although some improvements can—and perhaps should—be made, this is simply to improve upon existing functionality and make the system more efficient, less verbose and easier to use. The current version works exactly as it should according to the specifications and the task description.

Some testing of the system has been performed, but this is a very difficult task to automate since I have not found a similar system, or a combination of systems, that can easily create regular expressions of this kind, associate functionality with search path positions, test them with several inputs of different data types and predict and compare the results to the ones generated by the

PMatch library. Because of this, only the most obvious corner cases have been tested to find potential bugs. If errors in code should reveal themselves after more elaborate use of the library, anyone should feel free to make any changes to the code and/or contact the author at the email address *kvolden@gmail.com*.

The next step is now to use this library to try and prove that nondeterministic finite-state machines as a fundamental unit of design is possible. If it proves to be, and analyses show that this might be a good way to exchange cheap chip area into expensive flexibility and speed, one would start to look into creating more complex simulation models and how this could and should be implemented as a self cloning state machine on an actual FPGA.

Appendix A

SystemC listings

Note that the code uses some features from the upcoming C++ standard *C++11*, also known as *C++0x*, and needs to be compiled with the appropriate arguments. In the g++ compiler, this is done through the compiler argument '*-std=gnu++0x*'.

A.1 pmatch.h

```
#ifndef PMatch_h
#define PMatch_h

#include <string>
#include <deque>
#include <vector>
#include <functional>
#include <systemc.h>
#include "state.h"
```

```
#ifdef PMATCHLDEBUG
#include <iostream>
#endif

template <class Functor=std::function<void()>>
class PMatch : public sc_module{
private:
  std::deque<State<Functor>> states;

  int max(int a, int b){
    return (a < b) ? b : a;
  }

  int get_highest_callback_index(std::string reg_exp){  // Returns
      the highest functor index in the string
    int highest_index = 0, i = 0;

    while(i < reg_exp.length()){
      switch(reg_exp.at(i)){
        case '\\':
          i += 2;
          break;
        case '@':
          highest_index = max(highest_index, atoi(reg_exp.substr(++
              i).c_str()));
          break;
        default:
          i++;
          break;
      }
    }
    return highest_index;
```

```
}

std::string extract_subexpression(std::string expression, int
    start_pos){ // Returns subexpression, takes expression and
    start position of subexpression
  int i = start_pos + 1;
  int open_parentheses = 1;      // Used to count nested
      parentheses

  while(open_parentheses > 0){
    switch(expression.at(i)){
      case '\\':       // If character is an escaping backslash,
          skip next character
        ++i;
        break;
      case '(':
        ++open_parentheses;
        break;
      case ')':
        --open_parentheses;
        break;
    }
    ++i;
  }
  return expression.substr(start_pos + 1, i - start_pos - 2); //
      Return subexpression without surrounding parentheses
}

std::string extract_unit(std::string expression, int start_pos){
  int i = start_pos, end_pos = 0;

  while(end_pos == 0){
    ++i;
```

```
      if(expression.at(i) == '\\'){
        ++i;
      }
      else if(expression.at(i) == '>'){
        end_pos = i+1;
      }
    }
    return expression.substr(start_pos, end_pos-start_pos);
  }

  std::string build_quantified_subexpression(std::string expression
      , int min, int max){
    std::string built_expression = "";

    for(int i = min; i <= max; ++i){
      if(i > min){
        built_expression += "|";
      }
      for(int j = 0; j < i; j++){
        built_expression += expression;
      }
    }
    return built_expression;
  }

  void build_from_expression(std::string reg_exp, State<Functor> *
      start_state, State<Functor> *end_state, std::vector<Functor>&
       functions){
    enum {ONE, ZEROORMORE, ONEORMORE, ZEROORONE, OTHER} quantifier;
    int quant_min, quant_max, callback_n, current_pos = 0;
    State<Functor> *new_state, *from_state = start_state;
    std::string unit, subexpression;
    std::vector<State<Functor>*> loose_ends;
```

```
while(current_pos < reg_exp.length()){
  switch(reg_exp[current_pos]){
    case '<':                  // Marks start of unit
    case '(':                  // Marks start of subexpression
      if(reg_exp[current_pos] == '<'){     // If it's a unit
        unit = extract_unit(reg_exp, current_pos);
        current_pos += unit.length();
        subexpression.clear();
      }
      else{                    // If it's a subexpression
        subexpression = extract_subexpression(reg_exp,
            current_pos);
        current_pos += subexpression.length() + 2;  // +2 for
            parantheses stripped from subexpression
        unit.clear();
      }

      states.push_back(State<Functor>());
      new_state = &states.back();

      switch(reg_exp[current_pos]){
        case '*':
          quantifier = ZEROORMORE;
          ++current_pos;
          break;
        case '+':
          quantifier = ONEORMORE;
          ++current_pos;
          break;
        case '?':
          quantifier = ZEROORONE;
          ++current_pos;
```

```
      break;
    case '{':
      quantifier = OTHER;
      ++current_pos;

      quant_min = atoi(reg_exp.substr(current_pos).c_str())
          ;
      while(reg_exp[current_pos++] != ',');
      quant_max = atoi(reg_exp.substr(current_pos).c_str())
          ;
      while(reg_exp[current_pos++] != '}');

      if(!subexpression.empty()){    // If it's a
          subexpression
        subexpression = build_quantified_subexpression("("
            + subexpression + ")", quant_min, quant_max);
      }
      else if(!unit.empty()){      // If it's a unit
        subexpression = build_quantified_subexpression(unit
            , quant_min, quant_max); // Build a
            subexpression
        unit.erase();         // No longer a unit, but a
            subexpression
      }
      break;
    default:
      quantifier = ONE;
  }

  if(quantifier == ONE || quantifier == ONEORMORE ||
      quantifier == ZEROORONE || quantifier == OTHER){ //
      Add rules or build subexpression
    if(!unit.empty()){            // If it's a unit
```

```
      from_state->add_rule(unit, new_state);  // Add rule
    }
    else if(!subexpression.empty()){  // If it's a
        subexpression
      build_from_expression(subexpression, from_state,
          new_state, functions); // Run recursively
    }
  }

  if(quantifier == ONEORMORE || quantifier == ZEROORMORE){
    if(!unit.empty()){            // If it's a unit
      new_state->add_rule(unit, new_state); // Add rule
    }
    else if(!subexpression.empty()){  // If it's a
        subexpression
      build_from_expression(subexpression, new_state,
          new_state, functions);  // Run recursively
    }
  }

  if(quantifier == ZEROORONE || quantifier == ZEROORMORE){
      // Add epsilons
    from_state->add_epsilon(new_state);
  }


  if(reg_exp[current_pos] == '@'){  // If unit or
      subexpression has callback
    callback_n = atoi(reg_exp.substr(++current_pos).c_str()
        );
    new_state->add_functor(functions.at(callback_n));
  }
```

```
          from_state = new_state;
          break;

        case '|':
          loose_ends.push_back(&states.back());
          from_state = start_state;
          ++current_pos;
          break;

        default:
          ++current_pos;
      }

    }

    if(end_state != NULL){              // Connect all loose ends to
        end state
      for(int i = 0; i < loose_ends.size(); ++i){
        loose_ends.at(i)->add_epsilon(end_state);
      }
      from_state->add_epsilon(end_state);
    }
  }



public:
  PMatch(sc_module_name name) : sc_module(name){ }

  void initialize(std::string reg_exp, Functor *functor, bool
      continuous = true){
    initialize(reg_exp, std::vector<Functor>(functor, functor +
        get_highest_callback_index(reg_exp) + 1), continuous);
```

```
  }

  void initialize(std::string reg_exp, Functor functor, bool
      continuous = true){
    initialize(reg_exp, std::vector<Functor>(1, functor),
        continuous);
  }

  void initialize(std::string reg_exp, std::vector<Functor>
      functions, bool continuous = true){
    int callback_n;

    states.clear();
    states.push_back(State<Functor>());

    if(reg_exp.at(0) == '@'){
      callback_n = atoi(reg_exp.substr(1).c_str());
      states.front().add_functor(functions.at(callback_n));;
    }

    build_from_expression(reg_exp, &states.front(), NULL, functions
        );
    states.front().set_stay_enabled(continuous);

    states.front().mark_enable();

    for(int i = 0; i < states.size(); ++i){    // Update-enabled all
        to get all epsilon transitions from S0 too
      states.at(i).update_enabled();
    }
  }

  template <typename... Inputs>
```

```
  void step(Inputs... inputs){
    for(int i = 0; i < states.size(); ++i){
      states.at(i).step(inputs...);
    }

    for(int i = 0; i < states.size(); ++i){
      states.at(i).update_enabled();
    }
  }

  void restart(){
    for(int i = 0; i < states.size(); ++i){
      states.at(i).disable();
    }
    states.front().mark_enable();
    for(int i = 0; i < states.size(); ++i){   // Update-enabled all
        to get all epsilon transitions from S0 too
      states.at(i).update_enabled();
    }
  }

  bool is_empty(){
    bool contains_active_states = false;
    for(int i = 0; i < states.size(); ++i){
      contains_active_states = contains_active_states || states.at(
          i).is_active();
    }
    return !contains_active_states;
  }

  #ifdef PMATCH_DEBUG
  void debug_output_structure(){
    for(int i = 0; i < states.size(); ++i){
```

```
      std::cout << "S" << i << " (" << &states.at(i) << "):\n";
      states.at(i).debug_output_structure();
    }
  }
  #endif
};

#endif
```

A.2 state.h

```
#ifndef State_h
#define State_h

#include <vector>
#include <functional>
#include "rule.h"

template <typename Functor>
class State {
private:
  bool enabled, do_enable, stay_enabled, has_functor;
  std::vector<Rule<Functor>> transition_rules;
  std::vector<State<Functor>*> epsilon_transitions;
  Functor functor;

public:
  State(){
    enabled = false;
    do_enable = false;
    stay_enabled = false;
    has_functor = false;
```

```
  }

  void add_functor(Functor _functor){
    functor = _functor;
    has_functor = true;
  }

  void add_rule(std::string unit, State<Functor> *target_state){
    transition_rules.push_back(Rule<Functor>(unit, target_state));
  }

  void add_epsilon(State<Functor> *target_state){
    epsilon_transitions.push_back(target_state);
  }

  template <typename... Inputs>
  void step(Inputs... inputs){
    if(enabled){
      for(int i = 0; i < transition_rules.size(); i++){
        transition_rules.at(i).step(inputs...);
      }
      enabled = false;
      if(stay_enabled){
        mark_enable();
      }
    }
  }

  void mark_enable(){
    do_enable = true;
    for(int i = 0; i < epsilon_transitions.size(); i++){
      epsilon_transitions.at(i)->mark_enable();
    }
```

```
  }

  void update_enabled(){
    if(do_enable){
      enabled = true;
      if(has_functor){
        functor();
      }
    }
    do_enable = false;
  }

  void set_stay_enabled(bool _stay_enabled){
    stay_enabled = _stay_enabled;
  }

  void disable(){
    do_enable = false;
    enabled = false;
  }

  bool is_active(){
    return enabled;
  }

  #ifdef PMATCH_DEBUG
  void debug_output_structure(){
    for(int i = 0; i < transition_rules.size(); ++i){
      transition_rules.at(i).debug_output_structure();
    }
    for(int i = 0; i < epsilon_transitions.size(); ++i){
      std::cout << "---E--->" << epsilon_transitions.at(i) <<
          std::endl;
```

```
    }
    std::cout << "Has_callback_function:_" << ((callback_function)
        ? "Yes" : "No") << std::endl;
    std::cout << std::endl;
  }
  #endif
};

#endif
```

A.3 rule.h

```
#ifndef Rule_h
#define Rule_h

#include <string>
#include <vector>
#include "sub_rule.h"

template <typename Functor>              // Prototype for State
    pointers
class State;

template <typename Functor>
class Rule{
private:
  std::vector<Sub_rule> sub_rules;          // One sub_rule per
      input
  std::vector<char> bool_rels;             // Boolean relationships
      between the sub_rules
  State<Functor> *target_state;            // Pointer to transition'
      s target state
```

```
std::string unescape(std::string escape_string){  // Remove
    escaping backslashes
  for(int i = 0; i < escape_string.length(); i++){
    if(escape_string.at(i) == '\\'){       // Is escaped character
      escape_string.erase(i, 1);         // Delete escaping
          backslash, keep/skip escaped char
    }
  }
  return escape_string;
}

std::vector<std::string> split_fields_to_vector(std::string
    field_set){
  std::vector<std::string> split_fields;
  int start_of_field = 0;

  for(int i = 0; i < field_set.length(); ++i){  // Split the
      comma separated fields into a string-vector of fields
    if(field_set.at(i) == '\\'){
      ++i;
    }
    else if(field_set.at(i) == ','){
      split_fields.push_back(field_set.substr(start_of_field, i -
          start_of_field));
      start_of_field = i + 1;
    }
  }
  split_fields.push_back(field_set.substr(start_of_field)); //
      Last field

  return split_fields;
}
```

```
std::vector<std::string> split_if_range(std::string field){
  std::vector<std::string> range_from_to;

  for(int i = 0; i < field.length(); ++i){       // Search for not
      -escaped dash
    if(field.at(i) == '\\'){
      ++i;
    }
    else if(field.at(i) == '-'){
      range_from_to.push_back(field.substr(0, i));
      range_from_to.push_back(field.substr(i + 1));
    }
  }

  return range_from_to;
}

std::vector<bool> recursive_compare(int input_number){  // Empty
    function to end recursive function below
  return {};
}

template <typename First, typename... Rest>
std::vector<bool> recursive_compare(int input_number, First first
    , Rest... rest){
  std::vector<bool> results, append;

  results.push_back(sub_rules.at(input_number).match(first));
  append = recursive_compare(++input_number, rest...);
  results.insert(results.end(), append.begin(), append.end());
  return results;
}
```

```
void create_sub_rule_from_string(std::string sub_unit){   //
    Create the sub rule from the string representation
  std::vector<std::string> values, ranges, split_fields,
      new_range;
  bool invert = false;

  if(sub_unit == "."){                    // Wildcard
    sub_rules.push_back(Sub_rule());
  }

  else if(sub_unit.at(0) == '['){              // Several values/
      ranges in square brackets
    sub_unit = sub_unit.substr(1, sub_unit.length()-2); // Trim
        square brackets
    if(sub_unit.at(0) == '^'){                // If NOT-condition
      invert = true;
      sub_unit.erase(0, 1);
    }

    split_fields = split_fields_to_vector(sub_unit);  // Divide
        the string of fields into a vector of fields

    for(int i = 0; i < split_fields.size(); ++i){
      new_range = split_if_range(split_fields.at(i));
      if(new_range.size()){
        ranges.push_back(unescape(new_range.at(0)));
        ranges.push_back(unescape(new_range.at(1)));
      }
      else{
        values.push_back(unescape(split_fields.at(i)));
      }
    }
```

```
      sub_rules.push_back(Sub_rule(invert, values, ranges));
    }

    else{                              // Single value
      values.push_back(unescape(sub_unit));
      sub_rules.push_back(Sub_rule(invert, values, ranges));
    }

  }

public:
  Rule(std::string regexp_unit, State<Functor> *_target_state){
    int sub_unit_start = 0;
    target_state = _target_state;
    std::string stripped_unit = regexp_unit.substr(1, regexp_unit.
        length()-2);

    for(int i = 0; i < stripped_unit.length(); i++){     // Separate
         the parts (inputs/sub units) of the unit
      if(stripped_unit.at(i) == '\\'){           // If next
          character is escaped, skip it (can't be boolean operator)
        i++;
      }
      else if(stripped_unit.at(i) == '&' || stripped_unit.at(i) ==
          '|'){
        create_sub_rule_from_string(stripped_unit.substr(
            sub_unit_start, i-sub_unit_start));
        bool_rels.push_back(stripped_unit.at(i));
        sub_unit_start = i+1;
      }
    }
```

```
    create_sub_rule_from_string(stripped_unit.substr(sub_unit_start
        ));  // Last part of the string
  }

  template <typename... Inputs>              // Variadic templates = C
      ++0x
  void step(Inputs... inputs){               // Steps the rule,
      enables target if conditions are met
    bool condition_met = false;
    std::vector<bool> sub_results = recursive_compare(0, inputs...)
        ;
    std::vector<bool> anded(1, sub_results.front());  // Push first
         sub result onto "anded" vector

    for(int i = 0; i < bool_rels.size(); i++){     // Do all the AND
        -ing and save temporary results in "anded" vector
      if(bool_rels.at(i) == '|'){
        anded.push_back(sub_results.at(i+1));
      }
      else{
        anded.back() = anded.back() && sub_results.at(i+1);
      }
    }

    for(int i = 0; i < anded.size(); i++){       // Do the OR-ing
        between the AND-ed values
      condition_met = condition_met || anded.at(i);
    }

    if(condition_met){
      target_state->mark_enable();
    }
  }
```

```
  #ifdef PMATCH_DEBUG
  void debug_output_structure(){
    std::cout << "--- <";
    for(int i = 0; i < sub_rules.size(); ++i){
      sub_rules.at(i).debug_output_structure();
      if(i < bool_rels.size()){
        std::cout << bool_rels.at(i);
      }
    }
    std::cout << "> ---> " << target_state << std::endl;
  }
  #endif
};

#endif
```

A.4 sub_rule.h

```
#ifndef Sub_rule_h
#define Sub_rule_h

#include <vector>
#include <string>
#include "pcast.h"

class Sub_rule {
private:
  bool invert;                    // Whether to invert final answer
      (not-transition)
  bool wildcard;
  std::vector<std::string> single_values;
```

```
  std::vector<std::string> ranges;

public:
  Sub_rule(bool _invert, std::vector<std::string> _single_values,
      std::vector<std::string> _ranges){
    wildcard = false;
    invert = _invert;
    single_values = _single_values;
    ranges = _ranges;
  }

  Sub_rule(){                          // Constructor without arguments
      means wildcard
    wildcard = true;
    invert = false;
  }

  template <typename Input>
  bool match(Input input){
    bool match = wildcard;  // Ends up returning "true" (short
        circuits) if wildcard, depends on input if not

    for(int i = 0; i < single_values.size(); i++){  // Match any
        single value
      match = match || (input == pcast::cast<Input>(single_values.
          at(i)));
    }

    for(int i = 0; i < ranges.size(); i += 2){     // Match any
        range
      match = match || (input >= pcast::cast<Input>(ranges.at(i))
          && input <= pcast::cast<Input>(ranges.at(i+1)));
    }
```

```
    return (invert ? !match : match);        // Return and invert if
        appropriate
  }

  #ifdef PMATCH_DEBUG
  std::string debug_escape_characters(std::string chars){
    for(int i = 0; i < chars.length(); ++i){
      switch(chars.at(i)){
        case '-':
        case ',':
        case '\\':
        case '^':
          chars.insert(i, "\\");
          ++i;
          break;
      }
    }
    return chars;
  }

  void debug_output_structure(){
    if(wildcard){
      std::cout << ".";
    }
    else{
      std::cout << "[" << (invert ? "^" : "" );
      for(int i = 0; i < single_values.size(); ++i){
        std::cout << debug_escape_characters(single_values.at(i));
        if(i != single_values.size()-1){
          std::cout << ",";
        }
      }
```

```
      if(single_values.size() && ranges.size()){
        std::cout << ",";
      }

      for(int i = 0; i < ranges.size(); i += 2){
        std::cout << debug_escape_characters(ranges.at(i)) << "-"
            << debug_escape_characters(ranges.at(i+1));
        if(i != ranges.size()-2){
          std::cout << ",";
        }
      }
      std::cout << "]";
    }
  }
  #endif

};

#endif
```

A.5 pcast.h

```
#ifndef pcast_h
#define pcast_h

#include <sstream>
#include <string>

namespace pcast{
  template <typename Input>
  Input cast(std::string& value){      // For casting string value
      to input data type
```

```
    Input converted_value;
    std::stringstream stream;
    stream << value;
    stream >> converted_value;
    return converted_value;
  }

  template<>                  // Partial template specification:
      string
  std::string cast(std::string& value){
    return value;
  }

  template<>                  // Partial template specification: char
  char cast(std::string& value){
    return value.c_str()[0];
  }

  template<>                  // Partial template specification: char
  signed char cast(std::string& value){
    return (signed char)value.c_str()[0];
  }

  template<>                  // Partial template specification: char
  unsigned char cast(std::string& value){
    return (unsigned char)value.c_str()[0];
  }

  template<>                  // Partial template specification: char
  const char cast(std::string& value){
    return value.c_str()[0];
  }
```

```
template<>                  // Partial template specification: char
const signed char cast(std::string& value){
  return (signed char)value.c_str()[0];
}

template<>                  // Partial template specification: char
const unsigned char cast(std::string& value){
  return (unsigned char)value.c_str()[0];
}

template<>                  // Partial template specification: char
    *
char* cast(std::string& value){
  return (char*)value.c_str();
}

template<>                  // Partial template specification:
    signed char*
signed char* cast(std::string& value){
  return (signed char*)value.c_str();
}

template<>                  // Partial template specification:
    unsigned char*
unsigned char* cast(std::string& value){
  return (unsigned char*)value.c_str();
}

template<>                  // Partial template specification:
    const char*
const char* cast(std::string& value){
  return value.c_str();
}
```

```
  template<>                    // Partial template specification:
      const signed char*
  const signed char* cast(std::string& value){
    return (signed char*)value.c_str();
  }

  template<>                    // Partial template specification:
      const unsigned char*
  const unsigned char* cast(std::string& value){
    return (unsigned char*)value.c_str();
  }


}

#endif
```

Listings

List of Figures

List of Tables

Bibliography

[1] Moore, G.; *Cramming more components onto integrated circuits*, 1965.

[2] Moore, G.; *Progress in digital integrated electronics*, 1975.

[3] Svarstad, K. et al; *Non-Deterministic Finite State Machines as a Fundamental Unit of Design for Run Time Reconfigurable Systems*, 2011.

[4] Volden, K.; *A SystemC implementation of nondeterministic finite-state machines*, 2011.

[5] *IEEE 1666 Standard SystemC Language Reference Manual*, 2006.

[6] Rabin, M.O.; Scott, D.; *Finite Automata and Their Decision Problems*, 1959.

[7] Kleene, S.C.; *Representation of Events in Nerve Nets and Finite Automata*, 1956.

[8] *Perl Regular Expressions Documentation.*

[9] Thompson, K.; *Regular Expression Search Algorithm*, 1968.

[10] *C++0x Support in GCC: http://gcc.gnu.org/projects/cxx0x.html*

[11] *What's New in Visual C++ 2010: http://msdn.microsoft.com/en-us/library/dd465215.aspx*

[12] *Microsoft Connect answer by Visual C++ developer Jonathan Caves: https://connect.microsoft.com/VisualStudio/feedback/details/463677/support-variadic-templates*

[13] *ISO/IEC 14882:2003, Programming languages — C++, §14.7.3/2*, 2003.