The LTL Checker Plugins

a (reference) manual

HT de Beer, PCW van den Brand
H.T.d.Beer@student.tue.nl, P.C.W.v.d.Brand@tue.nl

Eindhoven, September 20, 2007

Contents

1 Introduction **3**

2 How to use the LTL Checker Plugins **4**
2.1 Introduction 4
2.2 Starting the LTL Checker Plugin 5
2.3 Starting the Semantic LTL Checker Plugin 6
2.4 Using the Template GUI 6
2.4.1 Choosing a formula 7
2.4.2 Providing values for parameters 7
2.4.3 Changing default values and deleting formulae 9
2.4.4 Settings 9
2.4.5 Running the LTL Checker 10
2.5 Viewing the Results 11
2.6 Exporting the Results 12
2.7 Reusing the results 12
2.8 Using the stand alone LTL Parser 13

3 LTL Language **14**
3.1 Introduction 14
3.2 The Environment in which the Language operates 14
3.2.1 A running Example 16
3.3 Definitions and Comments 16
3.3.1 Comments 16
3.3.2 Attribute Definitions 17
3.3.3 Renaming of defined Attributes 19
3.3.4 Formula Definitions 20
3.4 Formula calls 23
3.5 Comparisons 25
3.5.1 Standard comparisons 25
3.5.2 The string type operators and literals 27
3.5.3 The set type operators and literals 27
3.5.4 The date type literals 28
3.5.5 The number type literals and expressions 28
3.5.6 Concept sets 28
3.5.7 Examples 29
3.5.8 Parse errors 30
3.6 Propositional Logic 30
3.6.1 Not 31

3.6.2 And . . . 31
3.6.3 Or . . . 31
3.6.4 Implication . . . 32
3.6.5 Bi-implication . . . 32
3.6.6 Examples . . . 32
3.6.7 Parse errors . . . 32
3.7 Quantificational Logic . . . 33
3.7.1 Universal Quantification . . . 33
3.7.2 Existential Quantification . . . 33
3.7.3 Semantic Extensions to Quantification . . . 33
3.7.4 Examples . . . 34
3.7.5 Parse errors . . . 35
3.8 Linear Temporal Logic . . . 35
3.8.1 Nexttime . . . 35
3.8.2 Eventually . . . 36
3.8.3 Always . . . 36
3.8.4 Until . . . 36
3.8.5 Examples . . . 37
3.8.6 Parse errors . . . 40

4 LTL Grammar **41**
4.1 LTL File . . . 41
4.2 Propositions . . . 41
4.3 Literals . . . 42
4.4 Identifier . . . 42

Chapter 1

Introduction

This document is both a user manual for the LTL Checker plugins of the ProM framework and a reference manual for the LTL Language.

If you are only interested in how to use the plugins, just read Chapter 2. In that chapter, you will read how to start and use the various LTL Checker plugins. At the end of the chapter, you will see that results obtained from using the LTL Checker plugins can be used again in the ProM framework as you can use every log.

On the other hand, if you are interested in how to specify your own LTL properties, you will have to read chapter 3 too. That chapter explores all the details of the LTL Language, from defining attributes to creating complex LTL expressions. This chapter uses a simple running example to explain all the language elements.

The last chapter contains the grammar of the LTL Language. It can be used as a short summary of the chapter about the LTL Language.

Chapter 2

How to use the LTL Checker Plugins

2.1 Introduction

This chapter explains how to use the LTL Checker plugins. There are two different plugins which are grouped under the name "LTL Checker plugins". These plugins are the normal LTL Checker plugin and the Semantic LTL Checker plugin.

The main difference between the normal and the Semantic LTL Checker plugins is the fact that the Semantic version can make use of semantic annotations in the definition of a property and in the values given to parameters of a formula. For example, the Semantic LTL Checker understands properties such as "task is an instance of a concept C or any subconcept of C". In constrast, the normal LTL Checker only works on actual names (strings) used in the log file, such as "task name is equal to X".

Both the the normal and the Semantic LTL Checker plugins are analysis plugins. These analysis plugins can be used on any MXML or SA-MXML (sematically annotated) log file. Of course, the normal LTL Checker cannot make use of the semantic annotations and the Semantic LTL Checker does not offer more functionality than the normal LTL Checker when used on non-annotated logs.

Both LTL Plugins also use the same LTL properties (formulae) defined in LTL Template files. LTL Template files are files containing the specification of properties written in the special LTL Language (Chapter 3). By default, a template file with some default LTL properties is used when you first start a plugin. After starting the plugin, it is also possible to open template files with self-defined LTL properties.

In the next sections of this chapter, all steps taken and windows encountered while using one of the LTL Checker plugins are explained in full detail. The first plugin to be explained is the normal LTL Checker plugin.

2.2 Starting the LTL Checker Plugin

Figure 2.1: An opened log.

The LTL Checker Plugin can be started on *any* log. Logs can be opened by choosing *Open supported file...* in the *File* menu.

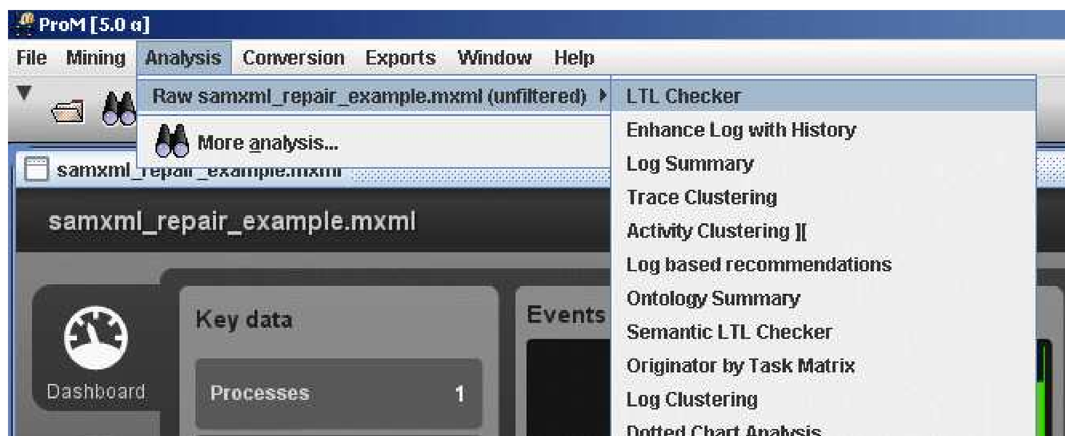

Figure 2.2: The analysis menu to start the LTL Checker Plugin.

Now, while the internal window with the log has focus, you can run the LTL Checker plugin via the menu *Analysis*, as is displayed in figure 2.2. After clicking on the LTL Checker menu item, a new window is opened: the Template GUI (Section 2.4).

Internally the standard LTL Template file is parsed. This file, called `standard.ltl` and can be found in the lib/plugins subdirectory of the ProM base directory. You can change the contents of this file to fit your own needs. If the standard file can not be opened or parsed, a message box with the error message is shown and the LTL Checker will not be started.

2.3 Starting the Semantic LTL Checker Plugin

The Semantic LTL Checker Plugin can be started on *any* log, but it's only useful to start it on a log file whose elements are semantically annotated: a so-called SA-MXML log file. SA-MXML logs can be opened like normal MXML log files by choosing *Open supported file...* in the *File* menu and they also have the extension .mxml.

Now, while the internal window with the log has focus, you can run the Semantic LTL Checker plugin via the menu *Analysis*, as is displayed in figure 2.2. After clicking on the Semantic LTL Checker Plugin menu item, a new window is opened: the Template GUI (Section 2.4). This is the same GUI as for the LTL Checker, except that there is now the option to select concepts for the parameter values.

The Semantic LTL Plugin also starts with the standard LTL Template file, and another file can be opened using the 'Open LTL file' button in the Template GUI.

2.4 Using the Template GUI

The Template GUI is a window consisting of several buttons and four main parts.

2.4.1 Choosing a formula

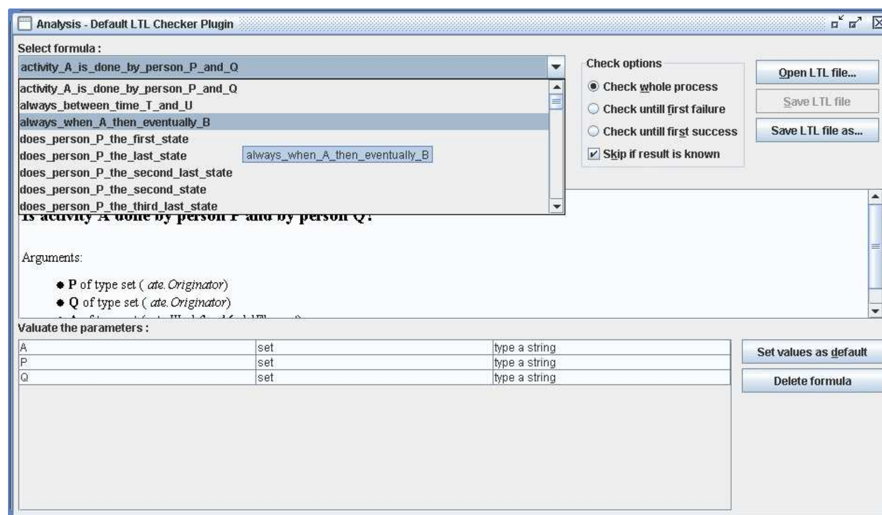

Figure 2.3: Choosing a formula in the Template GUI.

The first part is a list of formula names from which you can choose a formula. This selected formula is used to check on the log. Initially the first formula of the list is selected. If no formulae are defined the item *no formulae* is visible and selected.

The normal and Semantic LTL Checker plugins will always use the default formulae when you first start the plugin. If you want to use another formula than the ones provided by default, you can use the *Open LTL file...* button to open another file with LTL formulae. After selecting a file, the formulae will be parsed. If parsing is successful, the contents of the Template GUI will be updated to reflect the new file. If parsing was not successful, an error message will be shown and the file will not be opened.

In the second part of the window, the description pane, you will see the description of the selected formula (Figure 2.3). This description is created by the writer of the formula.

2.4.2 Providing values for parameters

Normal LTL Checker

The third part of the window is the parameter table. If the selected formula has any parameters these parameters are shown in this parameter table. It is up to you to fill in the values you want for the different parameters. Every row of the table exists of three columns: a name, a type and a value. Every parameter may have a default value, which can be specified in the formula definition (in the LTL file). If no default value is present for the formula, then the default values will be *0.0*, the date of today, or *type a string* for *number*, *date* and *string/set* parameters respectively.

You can only fill in those values that are acceptable values for the different kinds of attribute types. This means that every value, in case its type is number

or date, is parsed to a value of that type. If this parsing is successful, the new value is placed into the table, but if the parsing does not succeed the old value does not change.

It is important to know that you can only fill in literals. That is, attribute names, and other formulae are not accepted as values. They are interpreted as strings and, in cases of date and number attributes, they are parsed as dates or numbers respectively.

Semantic LTL Checker

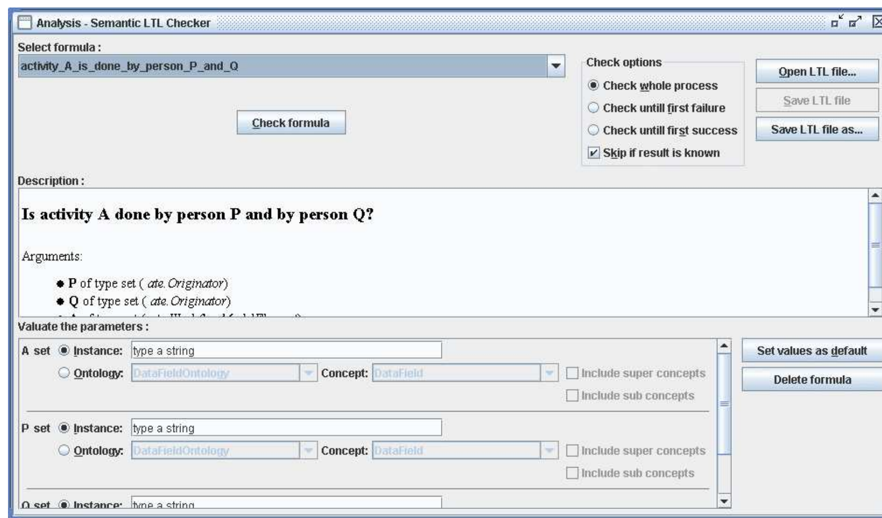

Figure 2.4: The Semantic version of the Template GUI.

In addition to using simple strings as a value for a parameter, the Semantic LTL Checker also supports *concepts* as a value (Figure 2.4) When a concept is selected, the LTL Checker will test whether the attribute is *an instance of* that concept, instead of testing whether the attribute value is equal to the given string. Concepts can be specified only for *set* attributes at the moment.

For example, suppose we have a formula which selects all cases where a certain task `A` is used. `A` is the only parameter and is a set of workflow model elements (tasks). In the normal LTL checker, one could give `A` the value `Send bill`, to select all cases in which the task with the name `Send bill` occurs. In the Semantic LTL Checker, one could select the concept `Billing` to select all traces which contain a task which is an instance of the `Billing` concept.

When the Semantic LTL Checker is started, it will load all ontologies which are referenced in the log file. The drop-down list 'Ontology' is filled with a list of all these ontologies. When the radio button next to 'Ontology' is selected, it is possible to choose an ontology from the list. When you choose an ontology, the 'Concept' list is filled with all concepts from that ontology, which are used in the log (including their super concepts).

If the checkbox 'Include super concepts' is checked, then also all superconcepts of the selected concept are included in the selection. For example, if

`Administrative task` is a super concept of `Billing` and 'Include super concepts is checked in the example above, then all traces are selected which contain a task which is an instance of `Billing` *or* `Administrative task`. Similarly, if 'Include sub concepts' is checked, then all subconcepts of the selected concept are included in the check.

2.4.3 Changing default values and deleting formulae

After changing the values of the parameters, you can choose to save these values as the default values for that formula. Clicking the *Set values as default* button will save the defaults only in memory. You can click the *Save LTL file* button to write the new defaults to the LTL file. The changes can also also be saved to another file using the *Save LTL File As...* button.

The *Delete formula* button can be used to delete the currently selected formula. Clicking this button will delete the formula from memory only; you need to use the *Save LTL file* button to make the changes permanent.

2.4.4 Settings

The fourth and final part contains some settings, and it is located to the right of the formula list. There are three options you can choose from:

1. *Check whole process* Check all process instances in the log for the selected formula.

2. *Check till first failure* Check all process instances in the log, but stop at the first encountered process instance **not having** the property specified by the selected formula.

3. *Check till first success* The reverse of the previous item. Check all process instances in the log, but stop at the first encountered process instance **having** the property.

The checkbox *Skip if result is known* below this list of options controls whether the LTL checker will re-check a formula it has checked before on the same log. When checking a formula, the LTL checker stores the result for each process instance (fail / succeed) in their data section. When this option is enabled and the result of the formula can be found in the data section, then the result is read from there. If this option is disabled, then the data section is ignored and the LTL checker will check the formula again (and write the result to the data section again).

2.4.5 Running the LTL Checker

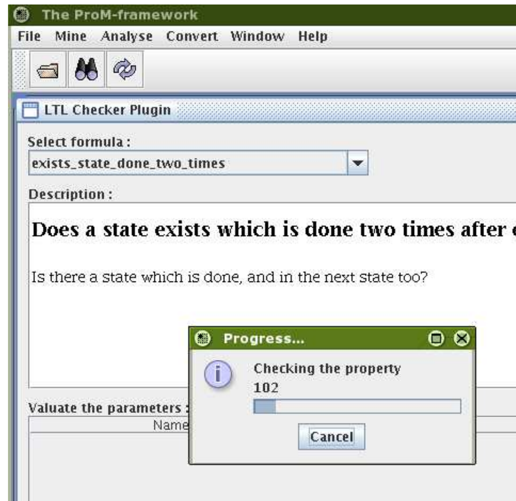

Figure 2.5: The actual checking is going.

If you have set the check option you want and you have provided the parameter values as you wish, you can click the *Check formula* button to start the actual check (Figure 2.5). While checking the formula, the progress of the check is displayed together with the name of the 'current' instance. If you want to stop the checking, just click the *Cancel* button on the progress window. After the check is done, a new window is displayed: the results window.

2.5 Viewing the Results

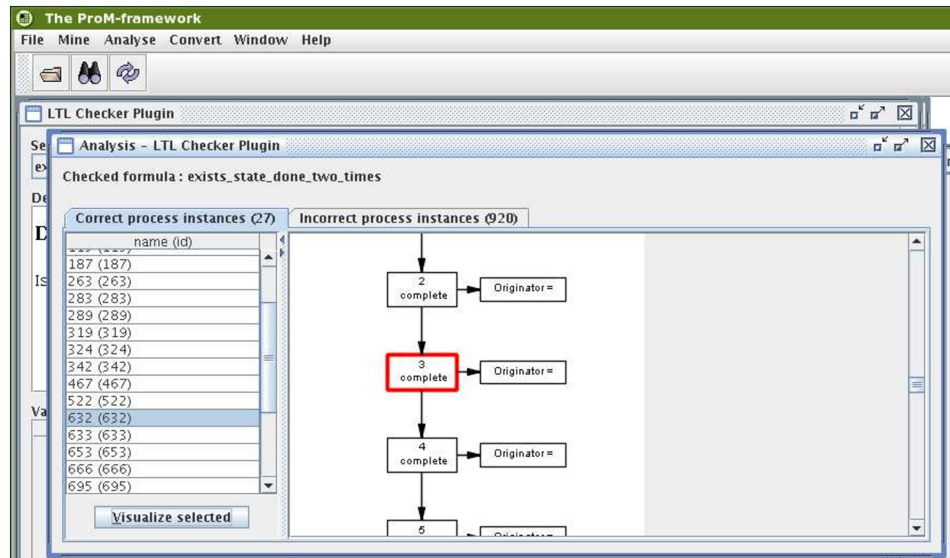

Figure 2.6: The window with the results: a tab with correct instances and one with incorrect instances.

On the results window you see first the name of the checked formula. Below this text, the window is divided into two tabs: one with the correct instances and one with the incorrect instances. Both tabs have the same structure. In the title of the tab, between parentheses the number of instances on that tab is given. So you can easily see how much instances of the log are correct or incorrect (Figure 2.6).

On the tab itself you see two parts: the actual table with the instances on the left and a visualization pane on the right. In this table, the process instances are listed, the name of the instances exists of the process instance name and between parentheses the number of times this instance occurs in the log.

The visualization pane on the right hand side of the log can be used to visualize one of the instances listed in the table. Just select an instance and click the *Visualize selected* button (or double-click in the table) to show the instance.

2.6 Exporting the Results

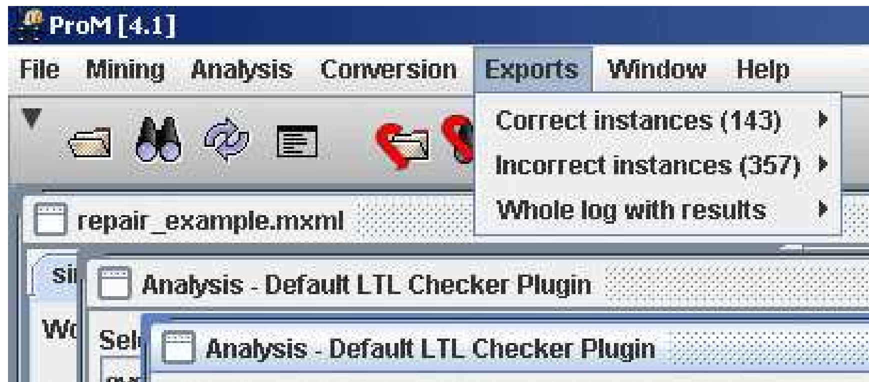

Figure 2.7: Exporting the results: correct and incorrect instances.

Both the lists of correct and incorrect instances can be exported or reused as ordinary logs can be. As can be seen in Figure 2.7, just go to the *Export* menu, and choose on the desired export. A save dialog is displayed then, in which you can give a name and a place to save the export to.

2.7 Reusing the results

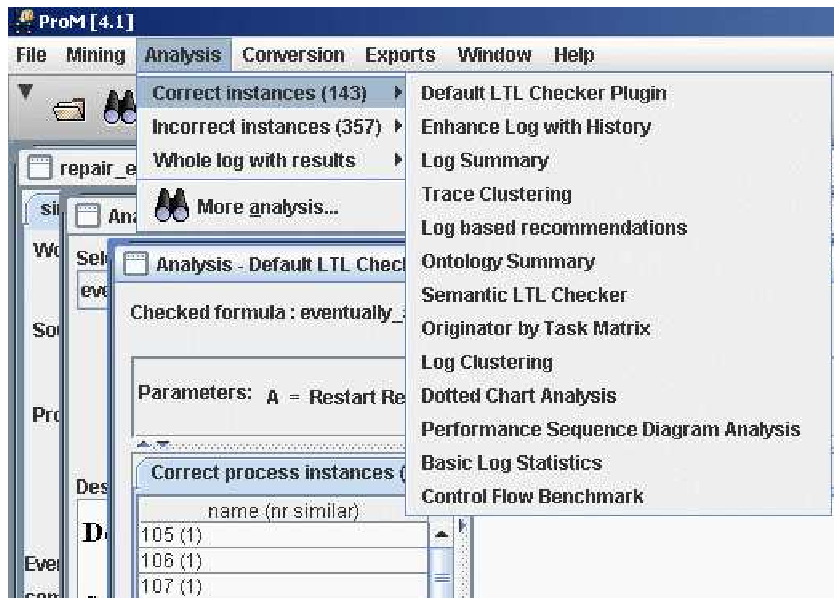

Figure 2.8: The LTL Checker (or any other plugin) can be run on the results again.

It is possible to use the results of a check again in the ProM framework, for example as input to the LTL Checker plugins. Both the LTL Checker plugin and the Semantic LTL Checker Plugin can be applied to a result log (Figure 2.8), either to the list with the correct instances or to the list with the incorrect instances.

So it is possible to refine logs as far as you want, until you are satisfied. For example, you can filter first all instances with property P, and then on those instances check for property Q. Of course, you can create a formula R equivalent to $P \wedge Q$, which gives the same results, but then you should know beforehand you want to check both P and Q. Furthermore, the smaller the log, the faster the check.

Not only the LTL Checker plugins can be used on the result logs, other plugins can be used too. For example the mining algorithms, or the analysis plugins.

2.8 Using the stand alone LTL Parser

The LTL Parser is also available as a stand alone program, so for debugging your own LTL Template files you do not need to start ProM. The stand alone LTL Parser can be started by typing the following on the command line, assuming you are in the base directory of ProM:

```
java -classpath "lib/plugins/ltlchecker.jar" ↩
org.processmining.analysis.ltlchecker.parser.LTLParser ↩
file_to_parse.
```

When you are using the Windows/DOS operating system, you have to change the slashes into backslashes. The LTL Parser expects exactly one filename.

If the parsing is successful, no message is send to standard out, the prompt returns directly. If there is an error while parsing, a message is send to standard out, consisting of the kind of error, and on the next line the message of the error. There are three kinds of errors possible:

1. `Error occurred during parsing:` The file is opened, but the contents are not parseble by the LTL Parser. The most likely reason is that you have made an error by writing ltl expressions.

2. `Error while reading` *`filename`*`. Check the file(name) and try again.` There are problems with opening and reading the file. Maybe the file does not exist or it is already in use by another program.

3. `Unknown error:` All errors not of kind one or two are unknown, that is, they are not expected by the parser but are thrown nonetheless. Most likely it is then a bug, so send a bug report containing the error message, and LTL Template file you try to import.

The stand alone program optionally accepts a -v parameter *after* the LTL file name. If this parameter is used and the file is successfully parsed, then the program will output the parsed file again (in another format). This is mainly useful for debugging purposes.

Chapter 3

The Language to specify LTL Properties: a reference

3.1 Introduction

Using the LTL Checker plugins with LTL Template files created by others is easy. Writing your own LTL Template files is another matter. This chapter tries to give a complete overview of the LTL Language, so that you can specify your own LTL properties.

Before the language itself is explained, the environment the language operates in is given.

3.2 The Environment in which the Language operates

The LTL Language specifies properties of event logs of processes. A process can be seen as a list of N process instances. Every process instance itself is a list of M ordered audit trail entries (Figure 3.1). Actual M can be different for any process instance. Both audit trail entries and process instances can have data fields. Audit trail entries always have a WorkflowModelElement and an EventType field, and may have other (optional) fields.

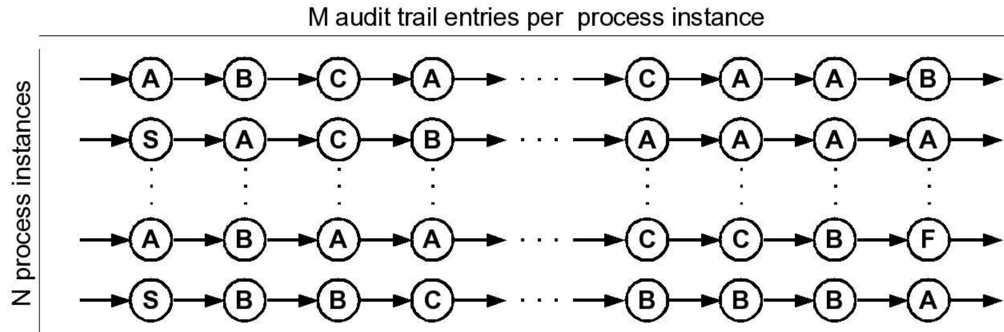

Figure 3.1: A log

In the LTL language those data fields can be accessed by defining (Subsection 3.3.2) attributes of the appropriate type (number, set, string or date) for the data fields. Later these attributes can be used in logical expressions via comparisons (Section 3.5).

The value of these attributes depends on the "current" audit trail entry while checking the log. A log is checked one process instance at a time. Every process instance is checked by inspecting its audit trail entries one by one. The current audit trail entry is that audit trail entry which is currently inspected by the checking algorithm.

The value of an attribute is computed by getting the value of the corresponding data field of the current audit trail entry. If needed, this string value is parsed to the appropriate type, thus in case the attribute is defined of type number or type date it is parsed. This ensures that comparisons work properly, i.e. 9 is less than 10 when comparing them as numbers, but "9" is larger than "10" when comparing them as strings.

In the language you can use different kinds of logical expressions. First of all, the "normal" propositional logic. This logic is about truth values only, that is, an expression is true if the operator applied on the operands results in true, otherwise it is false. The context of a propositional logic expression is determined by the temporal operators. If no temporal operators are used, a logical expression is only evaluated for the first audit trail entry of a process instance.

Another sort of logic is the quantificational logic which you can use to specify properties over all members of a set. Here, again, the context of a quantificational expression is determined by temporal operators.

The last kind of logical expressions are linear temporal logic expressions. With these LTL operators you can specify properties about the current audit trail entry, the next current trail entry given the current audit trail entry, any audit trail entry given the current or all audit trail entries given the current audit trail entry. The current audit trail entry is initially the first, so by not using temporal operators, all logical expressions are about the very first audit trail entry only.

The temporal operators are the operators which give you the ability to specify properties about a whole process instance, about all audit trail entries of a process instance. You can specify temporal relations between audit trail entries and so you change the context of the normal logical expressions.

3.2.1 A running Example

In this chapter we'll use a running example to clarify the LTL language constructs. Every process instance has two data fields: a name and a case number. For every audit trail entry at least four data fields are available: WorkflowModelElement (also called *task*), Originator (optional, also called *person*), Timestamp (optional, not used in this example), and EventType (which will always be *complete* in this example).

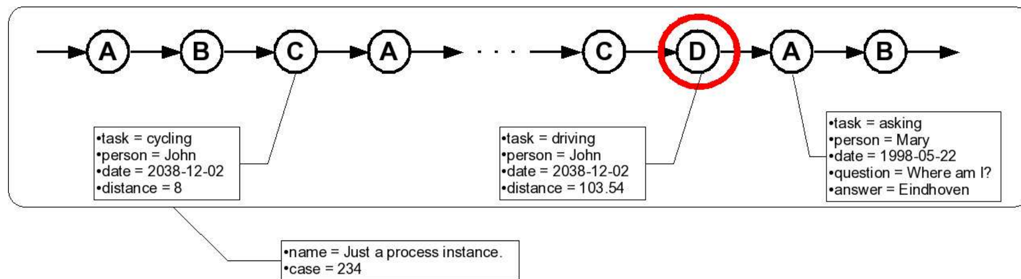

Figure 3.2: An example process instance

Besides those standard data fields, audit trail entries can have extra data fields depending on the task. In Figure 3.2 you see a visualization of one of such process instances of the example log. On the figure you see the current process instance, that is the process instance for which a property is computed. The current audit trail entry is the one with the red circle called **D**. When new language elements are introduced they are explained by using this figure.

3.3 Definitions and Comments

An LTL Template file can consist of four different items: comments, attribute definitions, renamings of defined attributes, and formula definitions. Those items can be mixed in any order, but an item can only be used after it is defined. Now the different items are described.

3.3.1 Comments

Comments are the most simple elements of the language. A comment starts with a `#`, and the rest of the line after the `#` (including the `#` itself) is skipped by the parser as comment.

It is a good practice to use comments in your LTL Template files to document your files in such a manner that you and others can understand and adapt the file later on, also after some time. It is very easy to create formulae, but to understand them can be more difficult if the formulae becomes more complex, then comments can be of great help.

Examples

We start our running example by creating some comment lines with some information about the file:

```
######################################
# version  : 0.0
# date     : 01122004
# author   : HT de Beer
##
```

Of course comments at the start of a file only is not that useful. It is also possible to use comments in definitions of formula, after attribute definitions, the line before renamings, and so on. Later on in this chapter, you will see examples of this.

Parse errors

No errors specified.

3.3.2 Attribute Definitions

An attribute definition starts with a keyword denoting the type of the attribute. There are four types and thus four keywords: `number`, `set`, `string` and `date`.

After the type you give the name of the attribute, followed by a semi-colon. The name must be unique and exists of two parts: a scope prefix and the name of the corresponding data element in the workflow log. The scope prefix is either "`ate.`" or "`pi.`" and denotes if the data element is an element of an audit trail entry (`ate.`) or if the data element is an element of a process instance (`pi`). So if a data element exists as well as a process instance element and as an audit trail entry element, you can define both as a separate attribute because the scope prefix results in two different names.

For a date attribute you should also use the `:=` keyword followed by a date pattern, and finally the semicolon. This date pattern describes the format of the dates in that particular attribute, so the LTL checker understands the date format. This is not needed for attributes which are not dates. Here is a more elaborate description of the different types:

- `number`
  A number attribute can contain values that can be parsed as a floating point number (including integers). Examples are: `12`, `12.34`, `12.34e5`, `9E-23` and `-123444`.

  You define a number attribute for data element *data-element* as follows: `number pi.data-element;` or `number ate.data-element;`, depending on the scope of the data element.

- `string`
  Attributes of type string are attributes which contain string values. Because in the log itself all data elements are strings, string attributes can be used for all data elements. No extra parsing is needed from the data element to an attribute. However, when you want to use special properties of date strings, number strings or sets, it is a good idea to define the data element to be of that type. For the string type, a special regular expression operator is available (Subsection 3.5.2).

You define a string attribute for data element *data-element* as follows: `string pi.data-element;` or `string ate.data-element;`, depending on the scope of the data element.

- `set`
  The set type is used for quantification, because quantification can only be used on set type attributes. For every set attribute a real set containing all the unique values of this data element of all process instances and all audit trail entries is created before checking the property. An extra set operator is the `in` operator (Subsection 3.5.3).

  You define a set attribute for data element *data-element* as follows: `set pi.data-element;` or `set ate.data-element;`, depending on the scope of the data element.

- `date`
  The last type is the date type. It can be used to access data elements containing date strings. Because there are many forms in which a date can be described, the date pattern is added to the date type attribute definition. The date pattern is afterwards used to create a SimpleDateFormat object of the Java programming language. The format of date patterns allowed is described at http://java.sun.com/j2se/1.4.2/docs/api/-java/text/SimpleDateFormat.html, a web page containing information about the SimpleDateFormat class.

  For now we give a short (incomplete) description on how to create a date pattern. For a full explanation, read the web page mentioned earlier. In short, a date pattern exists of characters with special meaning. If you want to put characters themselves in a data pattern, use a "'" to enclose them. Now a table with some date characters are given.

y	year
M	month in year
w	week in year
W	week in month
D	day in year
d	day in month
E	day in week
H	hour in day (0-23)
k	hour in day (1-23)
m	minute in hour
s	second in minute
S	millisecond

  With these date characters, you now can build date patterns, like:

  - *yyyy-MM-dd* : thus strings of the form 2006-12-09.
  - *'date='ddMMyy* : strings like date=230907.
  - *HH:mm* : strings denoting time like 12:45.

  You define a date attribute for data element *data-element* and date pattern *"date-pattern"* as follows: `date pi.data-element := "data-pattern";` or `date ate.data-element := "date-pattern";`, depending on the scope of the data element.

All data elements you want to use in your formulae should be defined as an attribute, including the standard data fields like WorkflowModelElement, EventType, Timestamp and Originator. These standard data fields are treated the same as user-defined data fields in the LTL Checker and LTL Language.

Examples

Back to our running example of figure 3.2. As you can see in this figure a number of attributes are available in the log. In this example we define all the attributes: ate.WorkflowModelElement, ate.Originator, ate.dob, ate.distance, ate.question, ate.answer, pi.name and pi.case.

```
#######################################
# version  : 0.1
# date     : 01122004
# author   : HT de Beer
##

##
# Defining attributes:
##
set ate.WorkflowModelElement;
set ate.Originator;
date ate.dob := "yyyy-MM-dd"; # Date of birth: dates have the format 'year-month-day'.
number ate.distance;
string ate.question;
string ate.answer;
string pi.name;
number pi.case;
```

Parse errors

- *Identifier is already defined*

  Occurs when you try to define an attribute with the same name as another earlier defined attribute.

3.3.3 Renaming of defined Attributes

Renamings make it possible to give a defined data element a shorter, more readable or more understandable name. In fact, you can define as many names for a data element as you want, as long as each name is unique.

Of course, as is usual in programming languages, a name can not be equal to one of the keywords. Thus `as`, `ate`, `date`, `exists`, `forall`, `formula`, `in`, `number`, `pi`, `rename`, `set`, `string`, `subformula` are not permitted as names for renamings and formulae. Furthermore, a name starts with a letter and can consist of letters, digits, -, and _.

A renaming can be created by the keyword `rename` followed by the attribute name, the keyword `as`, the new name and finally a semicolon. Thus for attribute name *old-name* and new name *new-name* you write a renaming as follows: `rename old-name as new-name;`.

Examples

We now adapt our running example so that the defined attributes are named as in figure 3.2. So ate.WorkflowModelElement is renamed as task, ate.Originator as person and ate.dob as bdate. Note that we do not use the scope prefix anymore in the new names.

```
#######################################
# version : 0.2
# date : 01122004
# author : HT de Beer
##

##
# Defining attributes:
##
set ate.WorkflowModelElement;
set ate.Originator;
date ate.dob := "yyyy-MM-dd"; # dates have the format 'year-month-day'.
number ate.distance;
string ate.question;
string ate.answer;
string pi.name;
number pi.case;

##
# Renamings
##
rename ate.WorkflowModelElement as task;
rename ate.Originator as person;
rename ate.dob as bdate;
rename ate.distance as distance;
rename ate.question as question;
rename ate.answer as answer;
rename pi.name as name;
rename pi.case as case;
```

Parse errors

- *Identifier is already defined.*

  The new name is already in use as another renaming or formula name.

- *Identifier is not a defined attribute.*

  You try to rename a formula or a renaming, but that is not possible. You can only rename attributes.

3.3.4 Formula Definitions

Now that you have defined attributes and have renamed them to something more readable, you can use those definitions to define formulae. There are two kinds of formulae: formulae and sub formulae. Both are defined exactly the

same but the first keyword, respectively `formula` and `subformula`, is different. The distinction is that sub formulae are not visible in the Template GUI (Section 2.4), that is, they can not be called by the user of the GUI. Formulae are the visible formulae, the formulae which the user can select and check.

A formula or subformula is defined as follows: the keyword `formula` or `subformula` followed by an unique name, a list of parameters between parentheses (the list may be empty), the keyword `:=`, a description, the actual formula and finally a semicolon. Thus: `formula A_formula_name ( arg1 : task, arg2 : person : "Default value" ) := { A description } actual formula doing something with the arguments ;`.

The different elements of a formula definition are now explained.

The list of arguments of a formula

The list of arguments is a comma separated list of so called local renamings, that is a new unique name in the local context followed by a colon and the attribute the new name is a local renaming for. The name of the local renaming can not be the same as a name of an attribute, renaming or formula, but it can be the same as an argument of another formula.

Furthermore the attribute, comparable with types in programming languages, is an already defined attribute (or a renaming of an attribute). Such a pair `argument_name : attribute_name` denotes that the name of the argument must be interpreted as a local renaming of the attribute. In expressions later on in the formula definition, this name can be used as such (but only on the right hand side of an comparison, see Section 3.5).

Optionally, each argument can be given a default value. This is denoted by adding another colon and then the default value after the attribute name. For example: `arg1 : task : "Send Bill"`. Note that this default value is *only* used as a default value in the Template GUI. It is *not* used as a default value when calling this formula from another formula (calling other formulas is explained below). Default values for `string`, `date` and `set` attributes should be quoted (e.g. `"some string"` or `"2000-01-01"`), while values for `number` attributes should be just a number without quotes (e.g. `1` or `3.14159`). Concept sets (Subsection 3.5.6) can also be used as default values for the Semantic LTL Checker.

The list, as said before, may be empty, but the parenthesis are obligatory in all cases, also if the list is empty.

The description of a formula

A description of a formula is the part of the formula definition in which you can inform the user of your defined formulae what the meaning and the use of the formula is. In the Template GUI (see section 2.4) the contents of the description are displayed in the description pane. Because the description pane renders its content as HTML 3.2, you can use those HTML 3.2 tags to create a more elaborate description for the user.

In fact, it is recommend to use HTML 3.2 code in your description. But only the contents of the body tag should be used because the rest of the HTML code is supplied by the plugin itself. Furthermore it is advisable to create your

descriptions along the guidelines given below, just to create a consistent view for the user.

A description is made out of curly braces where in between the contents of the description are placed. The guidelines for writing a description:

1. Start with the title, or name of the formula in between `<h2>` tags.
2. Divide your description into paragraphs using the `<p>` tags.
3. In the first paragraph you give a short description of the meaning of the formula in terms of the arguments.
4. in the next paragraph you list the arguments using an unordered list (using the `<ul>` tag). Every item in the list (create a list item with the `<li>` tag) starts with the name of the argument in bold (`<b>` tag), followed by the meaning of the argument. Then give the attribute and its type in italics (`<i>` tag). Then a recommendation of the value the user should give. For example a range or the pattern of the date pattern in case of a date attribute.
5. Write a more detailed description and/or other remarks if needed.

The actual formula

The actual formula is given in terms of propositional logic (section 3.6), quantification (section 3.7), linear temporal logic (Section 3.8), comparisons (Section 3.5) or (sub)formula calls (Section 3.4). That is, the actual formula consists of correct expressions (in this language) combined to larger expressions by the operators in this language.

Examples

For the running example we define now one formula, in which you see all elements of a formula definition. Because untill now you have not learned how to write the actual formula, a very simple formula is defined.

```
#######################################
# version : 0.3
# ...
# This part of the file is the same as in example version 0.2
# ...
##
# Formulae
##
formula eventually_task_A_is_done_by_person_P( A : task, P: person ):=
{
  <h2>Does P eventually do task A?</h2>

    <p>Is there a audit trail entry in a process instance in which person P
    does task A?</p>
```

```
    <p>
      <ul>
        <li><b>A</b> is a task, of attribute <i>ate.WorkflowModelElement</i>.
        For this argument fill in the task you want to check for.</li>
        <li><b>P</b> is a person, of attribute <i>ate.Originator</i>. Fill in
        the person's name for whom you want to know if he or she performed task A.</li>
      </ul>
    </p>
}
  <>( ( task == A /\ person == P ) );
```

Parse errors

- *Identifier already defined.*

  The name of this formula is already a defined identifier, either of an attribute, a renaming or another (sub)formula.

- *Identifier is already defined or used.*

  The identifier used as argument name is already used as argument name or as a global identifier, that is of an attribute, a renaming or a formulae.

- *Identifier is not a defined attribute or renaming.*

  The identifier used as 'type' of an argument is not a defined attribute or a renaming, so it can not fulfill the role of an argument type.

3.4 Formula calls

A defined formula can be called in the actual formula part of a definition of another formula. For calling a formula it is important that the formula you want to call must already be defined, you apply the right number of arguments, and, of course, of the right type. That is, an argument of type A must be applied with a value of type A, either an attribute itself (the same, modulo renaming) or a literal of the same type as the attribute.

You call a formula by entering the name followed by a parameter list surrounded by parenthesis. This list may be empty if the formula being called doesn't take any parameters.

The value of an argument in a call is bound at that place. With this feature, you get the value of a data element of an attribute of the audit trail entry that is current at the place the argument is used. If in the formula a nexttime operator is used (discussed below), the current audit trail entry becomes the next one, but the value of the argument supplied to this formula stays the of the current one.

This "early" binding can be useful when using the nexttime operator (Subsection 3.8.1), because then the current audit trail entry becomes the next one (if there is a next one of course). With a formula call, you can supply such nexttime operator with a value of a attribute before the then current. So you can apply recursion. You can read more on this subject in the subsection about the nexttime operator.

Examples

Because our running example has only one formula defined yet, we call it here in a new defined formula.

```
#######################################
# version : 0.4
# ...
# This part of the file is the same as in example version 0.3
# The formula eventually_task_A_is_done_by_person_P
# was also defined there.
# ...

formula does_John_drive() :=
{ Does John drive in a process instance? }
  eventually_task_A_is_done_by_person_P( "driving", "John" );
```

Note that concept sets can also be used as a literal when a `set` argument is required. For example:

```
formula does_employee_drive() :=
{ Does en employee drive in a process instance? }
  eventually_task_A_is_done_by_person_P(
          [ @http://example.org/Tasks#Drive ],
          [ @http://example.org/Originators#Employee ] );
```

Parse errors

- *Identifier is not a defined formula.*

  You used the identifier of an attribute or a renaming as the name of the formula to be called.

- *Defined number of parameters not equal the number of arguments applied here.*

  You tried to call a formula with the wrong number of arguments.

- *Identifier has not the right type.*

  You tried to use a identifier of an attribute or renaming or a local one with a different (attribute) type than defined in the formula definition. If at place x in the argument list a argument of type A is defined, use a value of type A on place x in the parameter list of the formula call.

- *Identifier is not a local parameter in this context.*

  You tried to call the formula with a identifier that is unknown in the local context, so it is not known as a defined attribute or renaming, and not as a argument to the formula in which the call is written down.

- *Unable to parse this as a date given definition xyz*

  You filled in a date string as parameter, but this string can not be parsed according to the definition of the used attribute in the argument list of the formula definition.

- *Type mismatch.*

  You tried to use a string value on an argument of attribute with type number.

- *Not expecting a integer.*

  A integer is applied where no number is expected.

- *Not expecting a floating point number.*

  A floating point number is applied where no number is expected.

3.5 Comparisons

The basis of formula definitions are the comparisons. Attributes (data elements of process instances or audit trail entries) can be compared with other attributes or with literal values. All attribute types have a number of "standard" comparisons common, and some have their own extra comparisons. First the standard ones will be explained, then the literals and special comparison operators are described per type. But before that, the common form of a comparison is given.

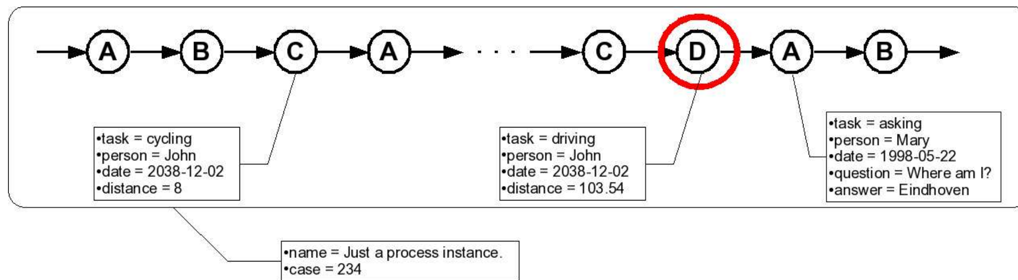

Figure 3.3: Again the example process instance

A comparison consists of a left hand side attribute (so not a local argument), with an operator and a right hand value. This right hand value is either a literal, an attribute (or renaming, or local argument) or a more complex expression in case of number attributes.

A comparison is always false if in the current audit trail entry the left hand side attribute does not exist. In the running example a comparison of the form `answer == "..."` results in false because the current audit trail entry, D, has no `answer` data element. So if you use a data element which does not exist for some audit trail entries, be aware of this feature.

3.5.1 Standard comparisons

`==`

The first standard comparison is the well known equivalence or "is equal" operator `==`. It compares the left hand side with the right hand side. It is true when both sides results in the same value given the current audit trail entry (and process instance).

In the example situation of Figure 3.3 where the audit trail entry labeled 'D' is the current audit trail entry, a comparison like `task == "driving"` would result in true, because `task`, referring to the `ate.WorkflowModelElement` and corresponding data element has the value `"driving"` in the log.

If the comparison was `task == "asking"` it would result in false, because the data element WorkflowModelElement of the current audit trail entry has value "driving".

In the Semantic LTL Checker, the right hand side of the `==` can be a concept set. If this is the case, then comparison is true when the attribute on the left hand side is an instance at least one of the concepts in the concept set on the right hand side.

Basically, the `==` operator is overloaded for concept sets to mean 'instance of'. In practice this means that the formula does not need to be changed if a user wants to compare the contents of the attribute (e.g. `task == "driving"`) in one case, or if a user wants to do a semantic comparison (e.g. `task == [ @http://example.org/Tasks#Driving ]`) in another case. This allows formulae to be used in more versatile ways and promotes re-use.

Only `==`, `!=` and `in` support concepts sets on the right hand side.

`!=`

The "is not equal" operator `A != B` is equal to `!( A == B )` (Subsubsection 3.6.1), and is the reverse of the `==` operator. So in the running example where D is the current audit trail entry, `task != "asking"` is now true and `task != "driving"` false.

This operator also supports concept sets on the right hand side. In this case it means 'the left hand side attribute is *not* an instance of one of the concepts on the right hand side'.

<=

The "lesser than or equal" operator is denoted as <= and does just what you expect that it does. Especially for date and number attributes this operator and the counterparts >=, < and > are useful. For string and set attributes you can use them for properties about the alfa numerical ordering of strings.

In the example `distance <= 200` results in true in the current audit trail entry there the distance is equal to 103.54.

>=

The bigger than or equal operator is denoted as >= and does just what you expect, as all the standard operators do.

<

The lesser than operator is denoted as <.

>

The bigger than is denoted as >.

3.5.2 The string type operators and literals

A string literal looks like string literals in most programming languages: two quotes in which between the contents of the string is written down. In principle every character can be in the string, only quotes themselves are not permitted because they close the literals. For example `"Hello world!"` or `"980u908009ud0f098s0d8uf0u8sd0ff8us09d"` are both correct strings, but `"I "like" quotes"` is not.

For strings there is an extra operator: the regular expression operator `~=`. This operator has as string literal a string which is interpreted as a regular expression pattern. Internally the *matches* method of the Java String class is used, so if you want to know all the details of the patterns have a look at http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html#sum.

Now follows a short introduction of regular expressions. With regular expressions you can specify patterns of characters, using special regular expression operators.

Operator	Meaning	Example
.	A arbritrary character	ma. : like may, mad, man, mau, ...
ˆ	The start of a line	
$	The end of a line	
regexp?	Zero or one time regexp	ma? : , ma
regexp+	One or more times regexp	ma+ : ma, mama, mamama, ...
regexp*	Zero or more times regexp	ma* : , ma, mama, mamama, ...
regexpx,y	Between x and y times regexp	ma1,2 : ma, mama
re1 | re2	re1 or re2	ma | pa : pa or ma
re1re2	re1 followed by re2	mapa : mapa, mamapa, mamamapa, ...
(re1)	Group re1	ma(pa)?ma : mama, mapama
[x-y]	Element of x...y	[1-9][0-9]+ : a number like 9, 83, 3859, ...
[abc]	Element of a b c	[dmy][1-9] : d3, y5, m5, m3, ...

With the building blocks given in this table you can construct more elaborate regular expressions, and more operators are available. To conclude this very short introduction on regular expressions, some examples:

- *.*word.** Search for strings with 'word' in it, like this sentence.
- *ID[0-9]{5,5}* Search for strings starting with ID and followed by five digits. ID98324, ID12347, etc.
- *a*b** Search for sequences of n a's, followed by m b's, like aabb, abbbbbbbb, aaaaaaaabbbb, etc.
- *yes|Yes|y|Y* Search for different ways to express yes.

3.5.3 The set type operators and literals

The set literals are exactly the same as string literals, because the sets consist of just strings. Sets are used for quantification, and therefore it is an different type.

One special set operator is defined: the `in` operator. It tests if the left hand side attribute is in a given set. It is of the form attribute, the `in` keyword, and a list of strings, separated by comma's between "[" and "]". It is ok to use

the same element more than once, although that will not influence the result of the `in` operator. In the running example an expression could be `person in [ "John", "Mary", "Angilbert"]`, that is, test whether the person of the current audit trail entry is either "John", "Mary", or "Angilbert". In this case, it results true because in the current audit trail entry 'D' the person is "John".

Using sets of strings as a parameter

The list of strings cannot be a parameter: it has to be a literal set of strings defined in the LTL formula. This restricts the usefulness of the `in` operator. As an alternative, it is also possible to use the `==` operator and rely on the feature of the Template GUI which splits strings on the `|` character into a set of strings. For example, suppose we define the following formula:

```
formula is_person_P( P: person ) :=
  person == P
```

Then in the Template GUI, you can enter the value `John|Mary|Angilbert` for the parameter `P` and the check will effectively become `person in ["John", "Mary", "Angilbert"]`. Note that strings are only split on the `|` character by the Template GUI, and not when they are used as literals in LTL formulae.

3.5.4 The date type literals

The date type has no extra operators, but the literals themselves are interpreted using the date pattern defined for the attribute on the left hand side of the operators. Those date patterns are explained in Subsection 3.3.2 above.

3.5.5 The number type literals and expressions

No extra comparison operators are defined for numbers, but more complex numerical expressions are possible. Number literals are either integers or floating point numbers. That is, just digits, or digits with a period and eventually an exponent (for example `123`, `0.4500988` or `314.9067E234`).

The numerical operators are the well known ones: `-`, `+`, `*` and `/`. The unary operator `-` is used as `- value` and the binary ones as `( value op value )`. The placing of parenthesis are a little bit strange and strict. Values can be either numerical expressions, number attributes (or renamings or arguments) or number literals.

3.5.6 Concept sets

A concept set defines a set of concepts in ontologies. These concept sets can be used on the right hand side of `==`, `!=` or `in` comparisons to test whether the attribute on the left hand side is an instance of one of the concepts defined in the concept set.

A concept set is denoted as a list of concept URIs between square brackets, where each URI is prepended with an `@` sign (e.g. `[ @http://ontology#Concept @http://example/another#One ]`). Only whitespace should be used to separate the elements, no commas.

Two special concept 'URIs' are defined to more easily include super concepts and/or subconcepts of a given concept. Use `@include-super-concepts` *directly after* the concept URI for which you want to include the super concepts, and use `@include-sub-concepts` *directly after* the concept URI for which you want to include the sub concepts. For example, the formula `person == [ @http://example.org#Employee @include-sub-concepts ]` will be true if the `person` is an instance of `@http://example.org#Manager` and the concept `@http://example.org#Manager` is a subconcept of `@http://example.org#Employee`.

The Template GUI of the Semantic LTL checker has support for selecting a concept using drop-down boxes, which makes it much easier to construct a concept set. The Template GUI of the normal LTL Checker has no such support and concept sets cannot be used as a value of a parameter.

3.5.7 Examples

To the running example some helper formulae are defined with using some comparison operators and literals. Because those comparisons are combined to larger expressions with logical operators (Section 3.6), those logical operators are used although the meaning may not be completely clear yet.

```
#######################################
# version : 0.5
# ...
# This part of the file is as in example version 0.4
# ...

subformula has_Answer() := {
 If the task is asking, then there is a answer not equal to the empty string.
 This formula results in true for all task other than asking and for those
 task equal to asking with a non empty answer.
}
  ( task == "asking" -> answer != "");

subformula distance_between( lbound: distance, ubound: distance ) := {
 The distance of the current audit trail entry lies between the lower bound
 and the upper bound. }
  ( distance >= lbound / distance <= ubound );

subformula reasonable_distance() := {
 If the task is cycling, the distance must be between 0 and 65, if the task
 is driving is must be between 0 and 300, if the task is flying, the distance
 must be between 150 and 1340. }
  (    ( task == "cycling" -> distance_between( 0, 65 )     ) /\
    (  ( task == "driving" -> distance_between( 0, 300 )    ) /\
       ( task == "flying"  -> distance_between( 150, 1340 ) )
    )
  );

subformula permitted_to_drive() := {
 A person is permitted to drive if he or she is born before 2004-6-01. }
```

```
  bdate < "2004-06-01";

subformula a_important_case() := {
 Case is important if the word 'important' is used in the name. }
  name ~= ".*important.*";
```

3.5.8 Parse errors

- *Identifier is not of type string.*

  This error is shown if you try to use the regular expression operator `~=` on attributes not of type string.

- *Identifier is not of type set.*

  This error is shown if you try the `in` operator with attributes not of type set.

- *Expected value of type number.*

  You have used the special numerical operators with values or attributes not of type number.

- *Unable to parse this as a date given definition . . . . .*

  Shown when you specify a date literal which is not parseble with the pattern specified by the associated left hand side date attribute.

- *Expected a string.*

  You use a left hand side attribute of type string, but the value you specified can not be parsed as such.

- *Identifier not defined.*

  You try to use an attribute or renaming as a value which is not defined before use.

- *Type mismatch*

  The left hand side attribute and right hand side value does not match types.

3.6 Propositional Logic

Now the comparisons and literals are explained, the basis for the propositional logic is laid because the atoms of the propositional logic in the LTL language are those comparisons. Let A and B correct atoms or formulae. They are used to combine them with logical operators to larger expressions.

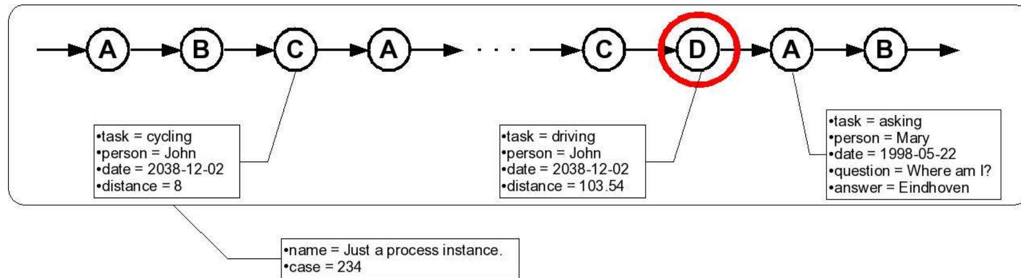

Figure 3.4: Again the example process instance

The explanation of the logical operators is done with the use of the running example where the current audit trail entry is as it was before: D (Figure 3.4).

3.6.1 Not

The proposition `!( A )` is true if and only if A itself is false. In a table:

A	`!( A )`
0	1
1	0

As an example let A be `answer == "Eindhoven"`. Now A is false, because there is no answer data element in the current audit trail entry D, so `!( answer == "Eindhoven")` is true. But the reverse holds too: let A be `distance < 1000`, with the distance equal to 103.54, you will see easily that the expression `!( distance < 1000 )` results in false.

3.6.2 And

The proposition `( A /\B )` is true if and only if A and B are together true. In a table:

A	B	`( A /\B)`
0	0	0
0	1	0
1	0	0
1	1	1

A simple example is `( task == "driving" /\person == "John" )` which is true. In the next audit trail entry, this would be false, both the task and the person are different, namely `asking` and `Mary`.

3.6.3 Or

The proposition `( A \/ B )` is true if and only if A or B is true (if both are true it is correct too, of course). In a table:

A	B	`( A \/ B)`
0	0	0
0	1	1
1	0	1
1	1	1

If the and operator in the last example is replaced with an or operator, it stays true in the current audit trail entry. It is even true in the third audit trail entry, where the person is `John` too.

3.6.4 Implication

The proposition `( A -> B )` is false if and only if A is true and B is false. But be aware, the arrow denotes a logical value, not a causal relation or a time relation at all. When one wants to state something about relations in time, thus that if A holds next time should hold B, use the nexttime operator `_O`. In a table:

A	B	(A -> B)
0	0	1
0	1	1
1	0	0
1	1	1

If a property is only important when another property holds, for example a (birth) date must be before "2000-06-05" when the task is `driving`. In this case, the implication operator is useful: `( task == "driving" -> bdate < "2000-06-05" )`. This can be read as: if the task is `driving`, then `bdate` should be less than `2006-06-05`.

3.6.5 Bi-implication

The proposition `( A <-> B )` is true if and only if A and B have the same value. This operator is also known as equivalence. In a table:

A	B	(A <-> B)
0	0	1
0	1	0
1	0	0
1	1	1

As an example we can state that a non empty question is equivalent with a non empty answer: `( question != "" <-> answer != "" )`. This formula can be read as 'if `question` is not empty, then `answer` should not be empty, and vice versa'. The 'and vice versa' is the difference with the implication operator (->). Thus, if both are not in the current audit trail entry, or both are empty, then it is true. If one of the data elements answer or question is empty or does not exist, then it is false. If both do exist and are not empty, it is true too. This is true in the current and the next state too.

3.6.6 Examples

In our running example we have already used some propositional logic operators, in the next two sections, the logical operators are also used, so for now no addition to the running example.

3.6.7 Parse errors

No errors specified.

3.7 Quantificational Logic

Quantification in this language consists of the universal (`forall`) and existential (`exists`) quantifier. They can be used to express that some formula should hold for all or for at least one element in a set, where a set is the set of all possible values of a `set` attribute. For example, `forall` can be used to express that a certain property should hold for all tasks or originators in a log.

The `forall` and `exists` operators are *not* meant to express properties that should hold for all or at least one audit trail entry in a process instance. For such properties, the temporal LTL operators `always` (`[]`) and `eventually` (<>) should be used. See below for more information on the temporal operators.

There is an important drawback to using quantification: it is very costly in terms of runtime, because quantification over a set S with property A results in a property of length |S| times the length of A. So it is advisable to try to use the LTL operators in stead of the quantifiers when possible. If you want to define a property on only one specific instance of a set, use the LTL operators.

The standard data elements of an audit trail entry, WorkflowModelElement, EventType and Originator are the most likely candidates for quantification, because they are usually defined as a `set` attribute. But all data elements can be sets and therefore used in quantifications. Even `Timestamp` or `distance` could be declared as `set`s and used in quantifications, but then the values would be interpreted as strings instead of dates or numbers.

3.7.1 Universal Quantification

The universal quantification computes if a property A holds for every element in a set S. That is the property $\forall_{s \in S}(A_s)$. An universal quantification is written as follows: `forall [ i: SetAttribute |` A_i `]`. With i: SetAttribute a local renaming i for attribute SetAttribute of type set. The property to check for all i is then A. If A holds for all i, this quantification is true.

3.7.2 Existential Quantification

The existential quantification computes if a property A holds for at least one element in a set S. That is the property $\exists_{s \in S}(A_s)$. An existential quantification is written as follows: `exists [ i: SetAttribute |` A_i `]`. With i: SetAttribute a local renaming i for attribute SetAttribute of type set. The property to check for all i is then A. If A holds for any i, this quantification is true.

3.7.3 Semantic Extensions to Quantification

When the log is semantically annotated, the elements in the sets used in quantification keep their model references. This means that the local renaming variable also carries semantic information (i.e. it knows which concept it is an instance of), so semantic checks can also be performed on the local renaming variable.

For example, it is possible to check whether all tasks in the log are instances of the `Task` concept:

```
formula all_tasks_are_instance_of_task_concept( ) :=
    forall [ t : task | t == [ @http://example.org#Task ] ]
```

3.7.4 Examples

In the running example we can now specify more complex properties using quantification. For example, a formula to test if there is a person doing two different tasks. The universal quantification is used to specify the property that all persons do either `moving` or `asking around`, not both.

```
#######################################
# version : 0.6
# ...
# This part of the file is as in example version 0.5
# ...

subformula P_does_A-A_not_B( P: person, A: task, B: task ) := {
 Compute if person P does task A, which is not equal to B.}
  <>( ( task == A /\ ( task != B /\ person == P ) ) );

formula exists_person_doing_two_different_tasks() := {
 Is there a person doing two different tasks?}
  exists[ p: person   |
    exists[ t: task   |
      exists[ u: task |
        (  P_does_A-A_not_B( p, t, u) /\
           P_does_A-A_not_B( p, u, t))
      ]
    ]
  ];

subformula moving( P: person ) := {
 Compute if person P is moving. }
  <>( ( person == P /\
        (   task == "driving" \/
          ( task == "cycling" \/
            task == "flying"
          )
        )
    ) );

subformula asking( P: person ) := {
 Compute if person P is asking. }
  <>( ( person == P /\ task == "asking" ) );

formula moving_or_asking() := {
 All persons are either moving or asking around.}
  forall[ p: person |
    (
      ( moving( p ) /\ !( asking( p ) ) ) \/
      ( !( moving( p ) ) /\ asking( p ) )
    )
  ];
```

3.7.5 Parse errors

- *Identifier already in use in the local context.*

  You tried to use a name for the dummy variable which is already used for something else.

- *Identifier is not a defined attribute or renaming.*

  The attribute you want quantify over is not defined.

- *Identifier has not type set.*

  The attribute you want quantify over is not a set.

3.8 Linear Temporal Logic

Again the figure of the running example, now to explain the LTL operators.

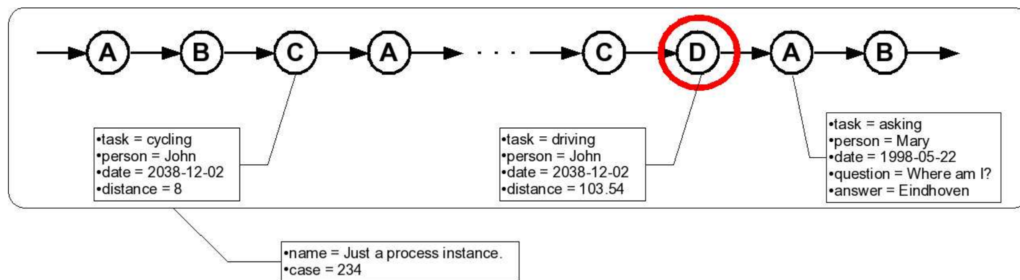

Figure 3.5: Again the example process instance

3.8.1 Nexttime

The nexttime operator is a very basic and simple operator. Given a current audit trail entry, it express something about the next audit trail entry in the process instance. So you can express properties about subsequent states.

The nexttime operator is a unary operator written as `_O( A )`. In Figure 3.5, where the current audit trail entry is D, `_O( person == "Mary" )` is true because in the next audit trail entry the person is `Mary`.

The nexttime operator can be used repeatedly to express some property about a specific audit trail entry given the current. For example, if the current audit trail entry was not D but C before it, then `_O(_O( person == "Mary" ) )` would be true too.

Use this `nexttime` operator to express recursion about states. Then, with the `always` operator (see below) you can check if a property A holds for all states of a process instance. Remark that a property `_O( A )` will never hold in the last state, so to apply such recursion you need to use something like '`always nexttime A` *or* `it is the last state`'. Combined with a formula call, you can use values of other states in the current comparisons.

For finalization, it is recommended to use an `end` formula:

```
subformula end() := {
 Only in the last state holds that in the next state a
```

```
 WorkflowModelElement is not equal itself (for other elements the same of
 course) because in the next audit trail entry of the last state all
 comparisons are false, since the data elements do not exist.}
  !( _O( ate.WorkflowModelElement == ate.WorkflowModelElement ));
```

Using this end formula, it is also possible to specify properties relative to the last state, if needed.

3.8.2 Eventually

The `eventually` operator checks if a property A occurs at least once in the sequence of audit trail entries of a process instance. With `<>( A )` you write an eventually expression down. Remark the correspondence with the `exists` quantifier, but this `eventually` operator is about a sequence of audit trail entries, and the `exists` quantifier about a set of values.

In the running example it is clear that `<>( person == "John" )` holds, there are audit trail entries with `person` is `John`, one of them the current audit trail entry. Given this current audit trail entry, the expression `<>( task == "asking" )` is true to, because the next audit trail entry is `asking`.

3.8.3 Always

`[]( A )` expresses that A always, so in all audit trail entries of a process instance, holds. This operator can be compared with the universal quantification, but again, the `always` operator is about a sequence of audit trail entries, and the universal quantifier about a set of values. This operator is true on the empty sequence or on the audit trail entry after the last one.

Use this operator to express that in all audit trail entries a property holds, for example that every next audit trail entry the date of birth is lesser or equal to the previous one. In this case the `end` formula is needed to finalize the formula, that is, to make sure that the property holds in the last state.

```
subformula next_older( d: bdate ) := {
 Is d bigger or equal the current date? }
  _O( bdate <= d );

formula always_nexttime_older() := {
 Holds always that the birth date of the person in he next audit trail entry is
 lesser or equal to the date of birth of the person performing the current
 audit trail entry?}
  []( ( next_older( bdate ) \/ end() ) );
```

3.8.4 Until

The `until` operator states that A holds until B holds. When B holds, A may or may not hold. You express this as `( A _U B )`. Remark that if B never holds and A always holds (from the current audit trail entry), the property is true.

As example a property expressing that until there is any distance moved, only questions are asked.

```
formula move_till_question() :=
    (task == "asking" _U  distance > 0  );
```

3.8.5 Examples

This section already contains useful examples, so here the last and complete version of the ltl file is given to conclude this chapter.

```
#######################################
# version : 1.0
# date : 01122004
# author : HT de Beer
##

##
# Defining attributes:
##
set ate.WorkflowModelElement;
set ate.Originator;
date ate.dob := "yyyy-MM-dd"; # dates have the format 'year-month-day'.
number ate.distance;
string ate.question;
string ate.answer;
string pi.name;
number pi.case;

##
# Renamings
##
rename ate.WorkflowModelElement as task;
rename ate.Originator as person;
rename ate.dob as bdate;
rename ate.distance as distance;
rename ate.question as question;
rename ate.answer as answer;
rename pi.name as name;
rename pi.case as case;

##
# Formulae
##

formula eventually_task_A_is_done_by_person_P( A : task, P: person ):=
{
  <h2>Does eventually P task A?</h2>

    <p>Is there a audit trail entry in a process instance in which person P
    does task A?</p>

    <p>
      <ul>
        <li><b>A</b> is a task, of attribute <i>ate.WorkflowModelElement</i>.
        For this argument fill in the task you want to check for.</li>
        <li><b>P</b> is a person, of attribute <i>ate.Originator</i>. Fill in
```

```
        the person you want to know if he or she perform task A.</li>
      </ul>
    </p>
}
  <>( ( task == A /\ person == P ) );

formula does_John_drive() :=
{ Does John drive in a process instance? }
  eventually_task_A_is_done_by_person_P( "driving", "John" );

subformula has_Answer() := {
 If the task is asking, then there is a answer not equal to the empty string.
 This formula results in true for all task other than asking and for those
 task equal to asking with a non empty answer.
}
  ( task == "asking" -> answer != "");

subformula distance_between( lbound: distance, ubound: distance ) := {
 The distance of the current audit trail entry lies between the lower bound
 and the upper bound. }
  ( distance >= lbound / distance <= ubound );

subformula reasonable_distance() := {
 If the task is cycling, the distance must be between 0 and 65, if the task
 is driving is must be between 0 and 300, if the task is flying, the distance
 must be between 150 and 1340. }
  (     ( task == "cycling" -> distance_between( 0, 65 )     ) /\
    (   ( task == "driving" -> distance_between( 0, 300 )    ) /\
        ( task == "flying"  -> distance_between( 150, 1340 ) )
    )
  );

subformula permitted_to_drive() := {
 A person is permitted to drive if he or she is born before 2004-6-01. }
  bdate < "2004-06-01";

subformula a_important_case() := {
 Case is important if the word 'important' is used in the name. }
  name ~= ".*important.*";

subformula P_does_A-A_not_B( P: person, A: task, B: task ) := {
 Compute if person P does task A, which is not equal to B.}
  <>( ( task == A /\ ( task != B /\ person == P ) ) );

formula exists_person_doing_two_different_tasks() := {
 Is there a person doing two different tasks?}
  exists[ p: person   |
    exists[ t: task   |
      exists[ u: task |
        (  P_does_A-A_not_B( p, t, u) /\
```

```
          P_does_A-A_not_B( p, u, t))
      ]
    ]
  ];

subformula moving( P: person ) := {
 Compute if person P is moving. }
  <>( ( person == P /\
        (   task == "driving" \/
          ( task == "cycling" \/
            task == "flying"
          )
        )
    ) );

subformula asking( P: person ) := {
 Compute if person P is asking. }
  <>( ( person == P /\ task == "asking" ) );

formula moving_or_asking() := {
 All persons are either moving or asking around.}
  forall[ p: person |
    (
      ( moving( p ) /\ !( asking( p ) ) ) \/
      ( !( moving( p ) ) /\ asking( p ) )
    )
  ];

subformula end() := {
 Only in the last state holds that in the next state a
 WorkflowModelElement is not equal itself ( for other elements the same of
 course) because in the next audit trail entry of the last state all
 comparisons are false, the data elements does obviously not exist.}
  !( _O( ate.WorkflowModelElement == ate.WorkflowModelElement ));

subformula next_older( d: bdate ) := {
 Is d bigger or equal the current date? }
  _O( bdate <= d );

formula always_nexttime_older() := {
 Holds always that the birth date of the person in he next audit trail entry is
 lesser or equal to the date of birth of the person performing the current
 audit trail entry?}
  []( ( next_older( bdate )  end() ) );

formula move_till_question() := {
 }
  ( task == "asking" _U  distance > 0  );
```

3.8.6 Parse errors

No errors specified.

Chapter 4

LTL Grammar

4.1 LTL File

```
<ltlfile>          ::= ( <attribute> | <renaming> | <formulae> )*

<attribute>        ::= ( ’number’ | ’string’ | ’set’ ) <attr_name> ’;’
                   |   ’date’ <attr_name> ’:=’ <date_pattern> ’;’

<date_pattern>     ::= <string_literal>

<attr_name>        ::= ( ’ate.’ | ’pi.’ )?<identifier>

<renaming>         ::= ’rename’ <attr_name> ’as’ <identifier> ’;’

<formulae>         ::= ( formula | subformula ) <identifier> ’(’ <arg_list>? ’)’
                               ’:=’ <desc_literal> <prop> ’;’

<arg_list>         ::= <arg> ( ’, ’<arg> )*

<arg>              ::= <identifier> ’:’ <attr_name> (’:’ <literal>)?
```

4.2 Propositions

```
<prop>             ::= <unary_prop>
                   |   <binary_prop>
                   |   <quantification>
                   |   <comparison>
                   |   <formula_call>

<unary_prop>       ::= ( ’!’ | ’[]’ | ’<>’ | ’_O’ ) ’(’ <prop> ’)’

<binary_prop>      ::= ’(’ <prop> ( ’/\’ | ’\/’ | ’->’ | ’<->’ | ’_U’ ) <prop> ’)’

<quantification>   ::= ( forall | exists ) ’[’ <identifier> : <name> ’|’ <prop> ’]’
```

```
<comparison>        ::= <attr_name>
                        ( '==' | '!=' | '<=' | '>=' | '<' | '>' | '~=' )
       <expr>
                    |   <attr_name> 'in' '[' <string_list>? ']'

<string_list>       ::= <string_literal> ( ',' <string_literal> )*

<formula_call>      ::= <identifier> '(' <param_list>? ')'

<param_list>        ::= <literal> ( ',' <literal> )*

<literal>           ::= <integer_literal>
                    |   <real_literal>
                    |   <string_literal>
                    |   '[' <concept_literal>* ']'

<expr>              ::= '-' <expr>
                    |   '(' <expr> ( '+' | '-' | '*' | '/' ) <expr> ')'
                    |   <literal>
                    |   <attr_name>
                    |   '[' <concept_literal>* ']'
```

4.3 Literals

```
<integer_literal>   ::= [1 - 9] ( [0 - 9] )*

<real_literal>      ::= ([0 - 9])+ . ( [0 - 9] )* <exponent>?
                    |   ( [0 - 9] )+ <exponent>?

<exponent>          ::= [e,E] ( [+,-] )? ( [0 - 9] )+

<string_literal>    ::= " ( ~[",\,\n,\r] | ( \ ( [n,t,b,r,f,\,',"] ) ) )* "

<desc_literal>      ::= { ( ~[{,}] | ( \ ( [n,t,b,r,f,\,',",},{] ) ) )* }

<concept_literal>   ::= @ (0-9,a-z,A-Z,~,!,@,#,$,%,^,&,*,(,),;,',:,,,.,/,?,-,+,=,_)+
```

4.4 Identifier

```
<identifier>        ::= <startletter> ( <letter> | <digit> )*

<startletter>       ::= [a - z,A - Z]

<letter>            ::= [a - z,A - Z,-,_]

<digit>             ::= [0 - 9]
```