Combining multiple similar grammars into a single modular grammar

Wijtse B. Rekker
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
w.b.rekker@student.utwente.nl

ABSTRACT

Implementing multiple similar grammars in one application often results in duplicate code and multiple parsers in general. This research discusses methods with which multiple similar grammars can be combined into a single modular grammar. In particular I take a look at the property languages used in the model checker LTSmin. Here, the similar languages CTL, LTL, and μ-calculus are used. They are implemented each with their own grammar. As a case study we determine if and how this can be improved with a modular grammar implementation.

Keywords

Modular grammar, Temporal logics, Similar grammars, Modular Grammar Converter

1. INTRODUCTION

In parser engineering, when multiple similar languages are used in a specific application, it is often easy to build individual grammars and parsers for each language even though they are similar. This might seem like a good solution, but it comes with some negative side effects. For example to use the tool, it has to be detected which parser should parse the input, which in turn adds more complexity. Another option is that the user specifies which parser should be used to parse the input. Side effects on the programming level are duplication of grammar rules and terminals, and overall more lines of code, which could lead to code which is more difficult to maintain. Therefore it can be highly beneficial to combine these similar grammars into one modular grammar, so there is only one parser that needs to be maintained.

This paper discusses different ways in how this can be achieved. In particular the model checking tool LTSmin[8] is taken as case study. LTSmin uses CTL*, LTL, state based μ-calculus, and action based μ-calculus as property checking languages. These are four languages with very similar properties, and they are each implemented with their own grammar. This provides a perfect example of the previously stated problem. In this paper I first discuss background information on some relevant topics, then the existing solution is evaluated, next the Modular Grammar Converter tool which I used for the new solution is explained, thereafter the new solutions are discussed, and lastly the validation of the solutions is discussed.

29th Twente Student Conference on IT July 6th, 2018, Enschede, The Netherlands.

1.1 Research questions

The main research question this paper answers is:

1) *"How can multiple similar grammars be combined into one modular grammar?"*

To help answer this question the following more specific question can be answered:

2) *"How is a modular grammar capable of parsing CTL*, LTL, and μ-calculus expressions best structured, and how does this improve on the previous implementation of LTSmin?"*

The following sub questions will help answer research question 2):

2a) *"How does the new solution affect the total amount of grammar rules?"*
2b) *"How does the new solution affect the total amount of tokens?"*

2. RELATED WORK

There already are a couple papers on modular grammar engineering, but not specifically about similar grammars. One of these papers is "Modular grammar specification"[7], by Adrian Johnstone, Elizabeth Scott, and Mark van den Brand. This paper discusses a modular approach to grammar building, and how to import syntax from other grammars. The problems in combining grammars that they focus on are: name hygiene, overlapping syntax, and separate whitespace handling per module. They propose semantics with which grammar modules can import nonterminals from other modules. Another paper in this area is "Modular Grammar Engineering in GF"[13], by Aarne Ranta. It discusses modular grammar engineering in the Grammatical Framework[12]. This is a grammar formalism specifically designed for multilingual grammars. Here the problem of duplicated code is solved with shared representations of multiple grammars. A different type of modular grammar engineering is introduced in "Compiler generation based on grammar inheritance"[1], by Mehmet Aksit, Rene Mostert, and Boudewijn R.H.M. Haverkort. Here modular grammars are specified with a mechanism similar to class structures in the programming language Java. Grammar rules can be inherited from super grammars like functions are inherited from super classes. Compared to the methods proposed by Adrian Johnstone et al. they provide a less fine grained way to import rules and productions from other grammars. Similar to LTSmin, CDAP[5]

is also an interesting model checking tool. Both tools use state based and action based μ-calculus. These languages are implemented in one grammar, alongside some extensions like regular expressions.

3. BACKGROUND

In this section some background information is provided on a couple topics relevant to this research. This information is required in order to understand some subjects further in the paper.

3.1 Modular grammars

Modular grammars are grammars built in modules. Here, modules typically are different parts of the grammar which can be changed without affecting the rest of the grammar as a whole. This can be useful when embedding a language in another language. This comes with some interesting challenges. For example when embedding SQL in the C language, the SQL comment brackets '{}' conflict with the scope brackets in C. Adrian Johnstone, Elizabeth Scott, and Mark van den Brand propose a method in their paper [7] which solves problems like these and more. They mainly propose extra syntax in addition to the usual grammar specification syntax. These extra rules can be used to specify grammars in a modular way. And with the semantics they propose, this modular grammar specification can be converted to a normal context free grammar specification. Since this paper will use this modular grammar specification, the syntax and semantics of the rules will be explained in the following section.

3.1.1 Syntax and semantics

Because the modular grammar specification syntax is an addition to the normal grammar specification syntax, the syntax still contains the normal rules like the following:

```
Module A;                Module B;
S ::= 'x' S | 'y' ;      S ::= 'xyz' ;
```

Here, module `A` is specified with one rule with on the left hand side the non terminal `S` and two productions on the right hand side. The non terminal `S` in module `A` and `B` are completely independent. Converting this to a normal grammar results in the following grammar:

```
A.S ::= 'x' A.S | 'y' ;
B.S ::= 'xyz' ;
```

Import by reference rule.

The import by reference rule is a rule specified in the form of `S <- M.T;`. This states that all the productions of the non terminal `T` in the module `M` are added to the productions of the non terminal `S`. As an example the following modular grammar is defined using the import by reference rule:

```
Module A;                Module B;
E <- B.F;                F ::= F '<' F
E ::= E '+' E                | '2' ;
    | '1' ;
```

Converting this to a normal grammar results in the following grammar:

```
A.E ::= A.E '+' A.E      B.F ::= B.F '<' B.F
      | '1'                    | '2' ;
      | B.F '<' B.F
      | '2' ;
```

The non terminal `A.E` now contains all the productions of the non terminal `B.F`. As is visible in the resulting grammar, the non terminal `A.E` can now parse for example: '`1+2<2`'. But it can not parse '`1<2`'. This problem is solved with the next rule.

Import by clone rule.

The import by clone rule is a rule specified in the form of `S <= M.T;`. This states that, much like the import by reference rule, all the productions of the non terminal `T` in the module `M` are added to the non terminal `S`. But every occurrence of `M.T` in these productions is replaced with `S`. As the example grammar we take the previous grammar, but with the import by clone rule instead of the import by reference rule.

```
Module A;                Module B;
E <= B.F;                F ::= F '<' F
E ::= E '+' E                | '2' ;
    | '1' ;
```

This converts into the following normal grammar:

```
A.E ::= A.E '+' A.E      B.F ::= B.F '<' B.F
      | '1'                    | '2' ;
      | A.E '<' A.E
      | '2' ;
```

Here you can see that the grammar is able to parse both '`1+2<2`' and '`1<2`'. A more complex example is now also possible: '`1+2<2+1<1`'. Even though the non terminal `B.F` is still included in the resulting grammar, it can not be reached when using `A.E` as start non terminal. So if `A.E` is selected as start non terminal of the final grammar, `B.F` can be removed from the result.

Import by clone recursive.

The import by clone recursive rule is an expansion on the rule discussed in the previous paragraph. It is specified in the form of `S <=* M.T;`. This does the same as the import by clone rule, but in addition it creates a statement `X <=* Y.X;` for every `Y.X` other than `M.T` in the productions of `M.T`. This is demonstrated in the following modular grammar:

```
Module A;                Module B;
E <=* B.F;               F ::= F '==' F
E ::= E '+' E                | X ;
    | E '-' E ;          X ::= N X | N ;
                         N ::= '0'
                             | '1'
                             | '2';
```

This results in the following normal grammar:

```
A.E ::= A.E '+' A.E      B.F ::= B.F '==' B.F
      | A.E '-' A.E            | B.X ;
      | A.E '==' A.E     B.X ::= B.N B.X
      | A.X ;                  | B.N
A.X ::= A.N A.X          B.N ::= '0'
      | A.N ;                  | '1'
A.N ::= '0'                    | '2';
      | '1'
      | '2' ;
```

When resolving the import rule '`E <=* B.F`' you also encounter the non terminal `B.X`, so a rule '`X <=* B.X`' is created in module `A`. While resolving this new rule recursively another import-by-clone-recursive rule is created for the non terminal `B.N`. Then this rule is resolved. Also in this case the rules of module `B` are not reachable from the rules of module `A`.

Remove production rule.

The last rule is a very simple rule. It is specified as `S :/= RHS;` where `S` is a non terminal and `RHS` could be any right hand side of a rule. This rule removes the production specified in the `RHS` part from the non terminal `S`. An example of this is given in the following grammar:

```
Module A;                Module B;
E <= B.E;                E ::= E '==' E
E ::= E '+' E                | E '&&' E
    | E '-' E                | 'true' ;
    | '1' ;
E :/= E '&&' E;
E :/= 'true';
```

Converting this into a normal grammar results in:

```
A.E ::= A.E '+' A.E      B.E ::= B.E '==' B.E
      | A.E '-' A.E              | B.E '&&' B.E
      | '1'                      | 'true' ;
      | A.E '==' A.E ;
```

The remove-production rules are always executed after the import rules. So the productions of `B.E` are first imported before the productions '`E '&&' E`' and ''`true`'' are removed from the non terminal `A.E`.

3.2 Temporal logic syntax

In this section the syntax of the temporal logics used in the property checker of LTSmin[8] is explained. The syntax that is shown here is not the actual syntax used by the tool, but the mathematical notation to get a better feel for the subjects and how their syntax are similar. CTL and LTL are inherently similar since they have a common ancestor logic CTL*. Temporal logics in model checking are generally used to specify properties that can be checked with traces of state machines.

3.2.1 CTL

CTL is short for *Computational Tree Logic*, and is a state based temporal logic, which means it can specify properties of states in traces of a state machine. Given a set **AP** containing *atomic propositions* the following can be expressed[3]:

Definition 1. Every atomic proposition $p \in \mathbf{AP}$ is a CTL formula. If f_1 and f_2 are valid CTL formulas, then so are:

$\neg f_1 \quad f_1 \wedge f_2 \quad EXf_1 \quad E[f_1 U f_2] \quad EGf_1$

3.2.2 LTL

LTL is short for *Linear Temporal Logic*, and is similarly to CTL also a state based temporal logic. Given a set of *atomic propositions* **AP**, the following can be expressed [4, 14]:

Definition 2. Every atomic proposition $p \in \mathbf{AP}$ is a LTL formula. If ϕ and ψ are LTL formulas, then so are:

$\neg\phi \quad \phi \wedge \psi \quad \phi \vee \psi \quad \phi \rightarrow \psi \quad X\phi$

$\phi \mathcal{U} \psi \quad \phi \mathcal{R} \psi \quad \Box\phi \quad \Diamond\phi$

3.2.3 mu-calculus

µ-calculus is used as well in model checking of state machines. Unlike CTL and LTL, µ-calculus additionally contains action based expressions. In other words, in µ-calculus formulas can be written to specify properties of states and transitions in traces in state machines. Transitions are also called *actions*. In order to help describe the syntax[2] a few sets are defined. Act is a set of all possible actions, S is a set containing all states, and $R_a \subseteq S \times S$ is a binary relation representing a transitions for every $a \in Act$. $Prop$ is a set containing all propositions, and a set $P_i \subseteq S$ exists for all $p_i \in Prop$ containing the states for which the proposition p_i holds. Var is a set containing all the defined variables. The formulas of the µ-calculus are expressed in the following definition:

Definition 3. Every proposition $p_i \in Prop$ is a µ-calculus formula, and every $X \in Var$ is as well. The rest of the µ-calculus are defined as follows:

- $\alpha \vee \beta$ and $\alpha \wedge \beta$ if α and β are µ-calculus formulas
- $\langle a \rangle \alpha$ and $[a]\alpha$ if $a \in Act$ and α is a µ-calculus formula
- $\mu X.\alpha$ and $\nu X.\alpha$ *if* $X \in Var$ and α is a µ-calculus formula

4. METHODS

To answer the research question with the LTSmin case study, a parser has to be made, satisfying LTSmin's needs and it needs to be a proper modular grammar. Inspiration will be taken from methods proposed in papers [7, 13]. Then the parser is tested and validated. For testing and validation unit testing will be used and inspiration will be taken from the methods proposed in [9, 11, 15]. The final product is compared to the previous implementation with special attention to amount of grammar rules, amount of tokens, lines of code, and parse time. The result of the evaluation of these metrics answers research questions 2a) and 2b). With these answers and the overall process, question 2) can be answered, and in turn question 1). The parser will be written in ANTLR[10], since this is the parser generator tool I am most familiar with.

5. EXPECTED RESULTS

It is expected the methods proposed in [7, 13] applied to the LTSmin case study result in a better grammar and parser. The expected results are that the final solution will have less grammar rules, tokens, and lines of code than the previous implementation. It is not expected that the parser will be significantly faster than the previous implementation.

6. EXISTING SOLUTION

The implementation of the property checking language parser in LTSmin[1] as of June 24 2018 is built in C with the open source LEMON Parser Generator[6]. The parser consists of three main components: the lemon lexer, the lemon grammar, and the run time environment.

The lemon lexer is located in the file `src/ltsmin-lib/ltsmin-lexer.l`. It only recognizes identifier, number, string, chunk, and operator tokens. Upon lexing an identifier or operator, it looks up the characters in tables filled by the run time environment. Based on these findings it parses different tokens of the grammar. This means that for example binary operators like '+' and '-' are not statically defined in the lexer or the grammar, but have to be put in a table of the parse environment on run time before the actual parsing takes place.

The lemon grammar can be seen as the base grammar of LTSmin. It is located in the file `src/ltsmin-lib/ltsmin-grammar.lemon`. For easier readability I translated the lemon grammar to a normal grammar which can be found in appendix A.1. This grammar is used for CTL*, LTL, and µ-calculus expressions, but it does not describe these languages as can be seen in the appendix. For example the

[1] `https://github.com/utwente-fmt/ltsmin`

BIN1-11 rules represent binary operators, but their character representations are not specifically defined. It is not specifically stated that, for example, BIN1 can be a '+' or a '-', and BIN2 a '*' or a '/'.

This is where the run time environment comes in. This is mainly located in the file src/ltsmin-lib/ltsmin-tl.c. While providing LTSmin with an input expression, the user has to specify which kind of expression it is. Then three different functions can be called. One for CTL* expressions, one for LTL, and one for μ-calculus. In these functions the tables of the parse environment are filled first, with the operators and tokens of the selected temporal logic. Then the provided expression is parsed, and the type checker is run with the result of the parse phase if no errors occurred. The effective grammars after the tables in the parse environment have been filled can be found in appendix A.2, A.3, and A.4.

6.1 Advantages

The advantages of this solution are:

- State and edge variable names in the expressions are checked on parse time, since the tables of the parse environment are also filled with the state and edge variable names. So the lexer will only lex correct variable names.
- The setup with the base grammar makes the operator priority definition easy. Because the BIN1-11 rules have fixed priorities of 1 to 11, and with the tables of the parse environment you can simply add operators to the production of their corresponding BIN non terminal.

6.2 Disadvantages

The disadvantages of this solution are:

- Since for every temporal logic the run time environment fills the tables of the parse environment separately and there is a fair amount of overlap between the temporal logics, several tokens and rules defined more than once. So if a feature is added, for example, to the predicate expressions, the same changes need to be made in multiple locations, which is extra work.
- Because there is not only one grammar with one start non terminal, but effectively three grammars, the user has to specify which grammar has to be used.
- In the current implementation all the grammars use only one grammar rule with a large amount of right hand sides. Because of this the grammar can parse some incorrect expressions like for example: '(2==4) +3'. This has to be fixed after the parsing phase by the type checker.
- In order to understand the parser you have to understand 4 fairly big code files, and how they work together. Because of this it is difficult to maintain the parser.

7. REQUIREMENTS

This section contains the requirements for the parser that is going to parse CTL*, LTL, and μ-calculus expressions. The requirements are written in the MoSCoW form.

Must.

These requirements must be met in order to get a minimal viable product.

1 The grammar must only have one start rule.

This is important, because it removes the need for the user to specify which kind of expression the user wants to parse.

2 The grammar must be able to parse CTL*, LTL, and μ-calculus expressions

Without this the grammar could not be used for LTSmin.

3 The parser must only accept expressions with the correct typing.

This means that for example an LTL formula can not consist of both LTL and CTL* expressions.

Should.

These requirements are also important, but not essential in order to get a working product.

4 The grammar must not contain any duplicate tokens.
5 The grammar should have less tokens than the previous grammar.
6 The grammar should have less grammar rules than the previous grammar.
7 The parser should be easier to maintain than the previous grammar.

All these requirements aim for a better solution than the current solution in LTSmin.

Could.

These requirements are focused on only if enough time is available.

8 The parser could maintain the correct operation ordering.

Since this is a difficult task to accomplish for the whole modular grammar, this requirement is placed in the 'Could' section.

Won't.

These requirements do not fit in the scope of this project, but are interesting for further research.

9 The parser won't be fully integrated in LTSmin.

8. MODULAR GRAMMAR CONVERTER

Since there are no tools that can read or convert modular grammar specifications, I built my own[2]. It is a tool with which you can convert any grammar written in the modular grammar syntax to a grammar with a normal syntax. Currently it is able to convert modular grammars to ANTLR4 grammars. The converter is built in such a way that it is possible to add more export modules, so it can be used to generate grammars for more parser generator tools tools.

The converter has four main stages. First it parses the modular grammar files and converts the parse tree to a abstract syntax tree. Next, error-checking is carried out on this abstract syntax tree. Then the modular grammar is converted to a normal grammar by resolving all the import rules, and lastly the normal grammar model is exported to an ANTLR4 grammar file.

8.1 Parse phase

For the parser side of the modular grammar converter ANTLR4[10] is used. The ANTLR grammar for the modular grammar syntax is included in appendix B. Because the Modular Grammar Converter is built with the inten-

[2]https://github.com/wijtserekker/ModularGrammarConverter

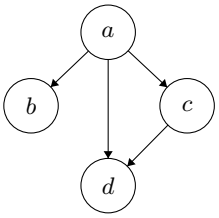

Figure 1. Example module dependency graph

tion to convert the modular grammars to ANTLR4 grammars, a lot of the features of ANTLR4 grammars are also embedded in the grammar used to parse modular grammar files, for example regular expressions and wild cards. The wild card token, however, is given the symbol '`$`' because the symbol used in ANTLR ('`.`') is already used for a different purpose. The regular expressions are implemented in a simple way (rule '`right_hand_side`' and '`regexp`' in appendix B), since the Modular Grammar Converter is not affected by incorrectly structured regular expressions, and the tool to which the modular grammar is exported to will already check the syntax of these expressions.

The parse tree created from the input is converted to an abstract syntax tree. This is an object oriented model of the modular grammar.

8.2 Error checking phase

The error checking phase is there to ensure that the program will not continue with incorrect input, and to make clear to the user why the input is incorrect. First it is checked if the parse tree contains any errors. Then the model is searched for the incorrect use and declaration of names. This can, for example, be two module names that are the same, the usage of a non-terminal from another module while not importing that module, or the usage of a non-terminal that does not exist. Then lastly the modular grammar is checked for cyclic module dependencies. The Modular Grammar Converter is not built to handle these complex dependency structures since it would take too much time to implement. In order to check this a dependency graph is made, and this graph is checked for cycles. For example the modular grammar below has the corresponding dependency graph shown in figure 1.

```
module a ;          module b ;
using b ;           ...
using c ;
using d ;
...

module c ;          module d ;
using d ;           ...
...
```

In this example the modular grammar has no cycles in the dependency graph, but if module `d` had a dependency on module `a` instead of `a` on `d`, then the dependency graph would be cyclic.

8.3 Conversion phase

To convert the modular grammar to a normal grammar, all the import rules have to be resolved. This needs to be done in a specific order. When for example module `a` uses module `b`, the import rules of module `b` have to be resolved before the module can be imported into `a`. Otherwise, module `a` does not get the complete module `b`. To get the right order of the modules, the module dependency graph is used. Traversing the dependency graph depth first returns the correct order of the modules. The module containing the start non-terminal is taken as the root of the graph. If we apply this, for example, to the dependency graph shown in figure 1 with as main module `a`, it results in the module order: `b`, `d`, `c`, `a`. And if the same is done, but with module `c` as main module, it results in the order: `d`, `c`. Here `a` and `b` are left out since they are not reachable from `c` and `d`, so these can be ignored during the conversion and export phase.

Now the module order is known, the import rules can be resolved. For every module the following steps are taken:

1. Generate import-by-clone rules for every import-by-clone-recursive rule.
2. Resolve all the other import rules.
3. Apply the remove-production rules.

This has to be done in this order. When, for example, a remove-production rule is applied. Then extra productions are added to the rule from which the remove-production rule tried to remove a production. This could result in the rule containing a production that the remove-production rule was supposed to remove.

When either an import rule or a remove-production rule is resolved, this rule is removed from the grammar model. This means that when the conversion has finished the model only contains normal grammar rules.

8.4 Export phase

In the export phase the grammar model is written to a single file using the ANTLR4 grammar syntax. Before the grammar is written to a file, a rule reachability check is run on the grammar starting from the main non-terminal. This checks which grammar rules are actually used in the final grammar. The grammar rules that can not be reached from the start non-terminal are not written to the output file.

The order of the rules and their productions is kept the same as the order in the modular grammar file. This is important since the order of the productions in a rule in ANTLR determine the priorities of the productions over each other.

To avoid overlap in non-terminal naming, the name of the module and the name of the rule are concatenated with a '`_`' in between. Also readability of the output file is kept in mind. Every rule of the grammar is printed on its own line and every production of a rule is also printed on a new line. The indentations used for the productions are the same as in the grammars shown in section 3.1.

9. NEW ARCHITECTURE

In order to create a modular grammar for the languages in LTSmin, first was looked at the overlap between the languages so the modules the whole grammar will consist of can be defined. As can be seen in the grammars in appendix A, each language uses the same predicate language. Therefore the predicate language is put in its own module. This can also be said for the boolean expressions, but not completely, because the language μ-calculus does not support the 'imply' and 'equivalent' operators. Also some temporal logic operators like the 'next' and 'exist' are used by multiple languages, so these tokens must also be moved to a module to prevent duplicate declarations.

From here there are two possible optimization paths in the way the modules can be linked together:

1. Reduce the rule and production count to a minimum.
2. Reduce the need for type checking.

These two paths go against each other because if you put all the possible expressions into one rule, the grammar parses a lot of unwanted expressions along the correct expressions. This is demonstrated in the following two grammars:

```
Grammar A;
E ::= E '+' E
    | E '-' E
    | E '<' E
    | E '>' E
    | N ;
N ::= 1 | 2 | 3 ;
```

```
Grammar B;
E ::= C '<' C
    | C '>' C
    | C ;
C ::= C '+' C
    | C '-' C
    | N ;
N ::= 1 | 2 | 3 ;
```

Here everything that grammar `B` is able to parse is also possible to parse with grammar `B`. In this example grammar `A` is also able to parse more expression combinations than grammar `B`, but these expressions are not correct, for example: '`1<3<2`'. Grammar `B` prevents this at the cost of one extra rule and one extra production.

In the following sections, solution 1 will aim to achieve the smallest amount of rules and right hand sides, and solution 2 will aim to prevent the need for type checking.

9.1 Solution 1

The modular grammar in this solution consists of 5 modules:

- *main* Containing the start non-terminal of the expression. It will import rules from the module *mucalc*, *ctl*, *ltl*, and *pred*. It combines all the modules into one rule.
- *mucalc* Containing the rule capable of parsing μ-calculus expressions. It uses the module *pred* for some token definitions.
- *ctl* Containing the rule capable of parsing all CTL* expressions. Like *mucalc* it also uses the module *pred* for some token definitions.
- *ltl* Containing the rule capable of parsing all LTL expressions. It uses the module *pred* for some token definitions.
- *pred* Containing all the possible predicate and boolean expressions. It does not use any of the other modules. It also contains some tokens used by multiple modules to prevent duplicate token definitions.

The dependency graph of the modules can be seen in figure 2. Since the syntax for the temporal logics are defined recursively (explained in section 3.2), they can be represented by one rule. For example the rule in the *ctl* module looks like this:

```
expr ::= 'E' expr
       | 'A' expr
       | '[]' expr
       | '<>' expr
       | 'X' expr
       | expr 'U' expr
       ;
```

This rule does not yet include the predicate expressions. Those will be added in the main module, because since these expressions are the same for the temporal logics they do not need to be imported in every module separately. To combine the expressions of all the temporal logics into one rule, the main module is written as follows:

```
module main ;
using ctl ;
using ltl ;
```

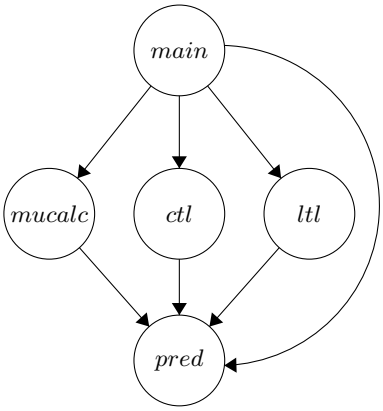

Figure 2. Dependency graph of the modular grammar in solution 1

```
using mucalc ;
using pred ;

expr <= ctl.expr ;
expr <= ltl.expr ;
expr <= mucalc.expr ;
expr <= pred.expr ;
```

The rule '`expr <= pred.expr;`' is added to also include the predicate expressions. This way the temporal logic expressions also have access to the predicate expressions. The final modular grammar of this solution can be found in appendix C.

9.2 Solution 2

The modular grammar in this solution consists of 6 modules:

- *main* Containing the rule which combines the temporal logics. It uses module *mucalc*, *ctl*, and *ltl*.
- *mucalc* Containing the rule which is able to parse complete μ-calculus expressions. It imports the predicate expressions from the module *pred*, and the boolean expressions from module *bool*.
- *ctl* Containing the rule which is able to parse complete CTL* expressions. Like the module *mucalc* it also uses the modules *pred* and *bool*.
- *ltl* Also works the same as modules *mucalc* and *ctl*.
- *pred* Containing all the predicate expressions, except for the boolean expressions. The boolean expressions are imported from the module *bool*.
- *bool* Containing all the boolean expressions.

The dependency graph of the modules can be seen in figure 3. The modules *mucalc*, *ctl*, and *ltl* are structured mostly the same as in section 9.1. The only difference is that here the predicate expressions are added already in the module itself instead of later in the module *main*. This is done with the following rules:

```
expr <= bool.expr ;
expr <= bool.expr_extra ;
expr ::= 'E' expr
       | 'A' expr
       | '[]' expr
       | '<>' expr
       | 'X' expr
       | expr 'U' expr
       | pred.comp_expr
       ;
```

The boolean expressions are added with the import-by-

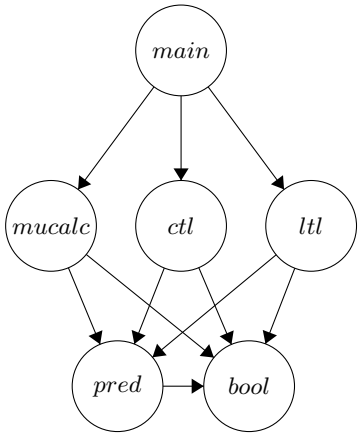

Figure 3. Dependency graph of the modular grammar in solution 2

clone rules. This makes it possible to parse expressions like '`(CTL expression)>(CTL expression)`'. To also include predicate expressions, the non-terminal `pred.expr` is added to the productions of `expr`. Importing predicate expressions like this makes sure that a CTL expression can not be a child of a predicate expression in the parse tree. The non-terminal `bool.expr_extra` contains expressions with the operators '`->`'(imply) and '`<->`'(equivalent). These operators are not supported in μ-calculus expressions in LTSmin currently, so they had to be imported separately.

The module *pred* also had to be structured in a special way to avoid the acceptance of expressions like '`(1==9)<2`'. As can be seen in appendix D, the predicate expressions have two layers: comparison expressions and calculation expressions. A smaller example of this structure is shown in the grammar below.

```
CO ::= CA '==' CA        CA ::= CA '+' CA
   | CA '<' CA ;             | CA '-' CA
N  ::= 1 | 2 | 3 ;           | N ;
```

The module *main* also has to preserve the type correctness, so in *main* a rule is created with three productions containing the start terminals for μ-calculus, CTL, and LTL.

```
expr ::= ctl.expr
     | ltl.expr
     | mucalc.expr
     ;
```

The complete modular grammar of solution 2 can be found in appendix D.

10. VALIDATION

To check for improvement on the existing solution the following elements are counted in the grammars:

- Rules. Includes normal grammar rules, token rules, and import rules. The lines at the top of modules specifying the module name and the imported modules are not counted.
- Productions. This is the total number of right hand sides of the rules of the grammar. Import rules are not counted here.
- Tokens. The total amount of token rules.
- Duplicate tokens. The total amount of duplicate token definitions. If for example a token is defined three times, it will count as two duplicate tokens.

Table 1. Comparison of the two modular grammars with the existing solution

	Rules	Prod.	Tokens	Dup. tokens
Existing sol.	137	223	137	65
Solution 1	49	84	41	0
Solution 2	54	92	41	0

Table 2. Comparison of the two generated grammars with the existing solution

	Rules	Prod.	Tokens	Dup. tokens
Existing sol.	137	223	137	65
Solution 1	42	78	41	0
Solution 2	47	108	41	0

These elements are counted in the existing solution, the modular grammar of solution 1, the modular grammar of solution 2, the generated grammar of solution 1, and the generated grammar of solution 2. Since the grammars of LTSmin are not setup in a normal way but with a base grammar, I do not simply count everything in appendix A. The parts that are included are in appendix A.1 until the rule '`CONSTANT ::= IDENT;`', in appendix A.4 after the rule '`VALUE ::= STRING | CHUNK;`', in appendix A.2 after the rule '`VALUE ::= STRING | CHUNK;`', and in appendix A.3 after the rule '`VALUE ::= STRING | CHUNK;`'. The results of the modular grammars compared to the existing solution can be found in table 1, and the results of the generated grammars compared to the existing solution can be found in table 2. At first glance it is visible that the numbers of the new solutions are significantly lower than those of the existing solution.

With this can be checked if the requirements specified in section 7 are met in the new solutions, except for requirement 2 and 3. These requirements are checked with Unit testing[3]. Here is tested if the grammars can actually parse the correct expressions, and that they fail where they are supposed to. Of the *must* requirements, all the requirements are met.

1 Because both solutions have only one start rule.
2 Because the Unit tests were successful.
3 Because the Unit tests were successful for solution 2. For solution 1 however the correct typing was not preserved, but this was as expected. This could be fixed by adding type checking after the parse phase.

All the *should* requirements are also met. This can be concluded by simply looking at table 1 and 2. Requirement 7 however is a difficult one to actually measure, but it is safe to say that understanding the grammars of the new solutions is a lot easier than the grammars of LTSmin. Because, in the new solutions you only have to look at the compact modular grammar file, and in LTSmin you have to go through 4 files. Requirements 8 and 9 are not met, but this was as expected. That was due to the limited amount of time available.

11. CONCLUSION

Overall, can be concluded from the results presented in section 10 that solution 2, presented in section 9.2, is overall a good solution for the temporal logic parser in the model checking tool LTSmin. It is only lacking the operator priorities. The amount of grammar rules, productions, and tokens are significantly reduced compared to the ex-

[3] `https://github.com/wijtserekker/LTSminGrammarTest`

isting solution. Now with this said the research questions can be answered.

The answer to question 2b) is, that the amount of tokens can be reduced significantly. All the duplicate token definitions have been eliminated along with some unnecessary token definitions. So the new solution affects the total amount of tokens positively. The answer to question 2a) is similar. The amount of grammar rules has been reduced significantly, so the new solution affects the total amount of grammar rules also positively.

To answer question 2), the structure of the modular grammar of the solution 2 is taken. A modular grammar capable of parsing CTL, LTL, and μ-calculus expressions should consist of the 6 modules *main*, *ctl*, *ltl*, *mucalc*, *pred*, and *bool*. The grammar rules should also be layered as described in section 9.2. This prevents the need for type checking the parse tree, which is necessary in the current solution in LTSmin. Along with the reduction of the size of the grammar, another improvement upon the current solution is that the grammar is more self contained compared to the existing grammar which was ultimately specified by four different files.

With the answer to question 2), the main research question 1) can be answered. When combining similar grammars into one modular grammar, the biggest part comes down to defining the modules correctly. The complete separate parts of the grammars should be defined in their own module (like CTL, LTL, and μ-calculus in the modular grammars of LTSmin), and when two or more parts overlap, those should be combined into one module which other modules can use. How these modules interconnect depends on the applications needs. In the case of LTSmin, the modules and rules had to be layered to preserve the correct formula typing.

12. FURTHER WORK

Since some features were not possible to implement, because of the limited time available, there are a few points that could be interesting for future research.

In the Modular Grammar Converter it is currently not yet possible to specify the priority of certain productions over other productions, especially for the productions imported from other rules since they are simply added at the end of the already existing productions. Another feature that can be added is support for cyclic dependencies. This is possible to achieve, but a fair amount of research has to be done to find a way how this can be implemented efficiently. Also multiple export modules could be added to the Modular Grammar Converter so the tool can be used for more parser generators.

Because the Modular Grammar Converter does not yet support the specification of the priorities of productions, the modular grammars designed for LTSmin do not preserve the correct operator priorities. This could be further developed so the modular grammars can be integrated in LTSmin.

13. ACKNOWLEDGEMENTS

Above all, I would like to thank my supervisor Jaco van de Pol for his guidance and great advice. Secondly, I would like to thank Jeroen Meijer for helping me understand the back-end of LTSmin. Also the feedback of the other people who reviewed my paper, sometimes more than once, was invaluable to the research. And lastly, many thanks to Adrian Johnstone, Elizabeth Scott, and Mark van den Brand for their research on the syntax of modular grammar specification used in this paper.

14. REFERENCES

[1] M. Aksit, R. Mostert, and B. Haverkort. Compiler generation based on grammar inheritance. *Memoranda informatica*, 0(07):–, 1990.

[2] J. Bradfield and I. Walukiewicz. The mu-calculus and model-checking. *Handbook of Model Checking. Springer-Verlag*, pages 35–45, 2015.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, Apr. 1986.

[4] E. A. Emerson. CHAPTER 16 - temporal and modal logic. In J. v. Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 995 – 1072. Elsevier, Amsterdam, 1990.

[5] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, Apr 2013.

[6] Hipp, Wyrick & Company, Inc. (Hwaci). The LEMON parser generator. `https://www.hwaci.com/sw/lemon/` (last visited 29-6-2018).

[7] A. Johnstone, E. Scott, and M. van den Brand. Modular grammar specification. *Science of Computer Programming*, 87:23 – 43, 2014.

[8] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-performance language-independent model checking. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 692–707, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[9] R. Lämmel. Grammar testing. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, pages 201–216, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[10] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[11] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, Sep 1972.

[12] A. Ranta. Grammatical framework. *Journal of Functional Programming*, 14(2):145–189, 2004.

[13] A. Ranta. Modular Grammar Engineering in GF. *Research on Language and Computation*, 5(2):133–158, Jun 2007.

[14] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In D. Bošnački and S. Edelkamp, editors, *Model Checking Software*, pages 149–167, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[15] M. Stijlaart and V. Zaytsev. Towards a taxonomy of grammar smells. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 43–54, New York, NY, USA, 2017. ACM.

APPENDIX

A. THE OLD GRAMMARS OF LTSMIN

A.1 The base grammar for the temporal logics

```
expr ::= LPAR expr RPAR
     | STATE_VAR
     | IDENT
     | EDGE_VAR
     | VALUE
     | NUMBER
     | CONSTANT
     | expr BIN1 expr
     | expr BIN2 expr
       ...
     | expr BIN11 expr
     | PREFIX1 expr
     | PREFIX2 expr
       ...
     | PREFIX9 expr
     | expr POSTFIX1
     | expr POSTFIX2
       ...
     | expr POSTFIX9
     | MU_SYM IDENT DOT expr
     | NU_SYM IDENT DOT expr
     | EXISTS_SYM expr DOT expr
     | ALL_SYM expr DOT expr
     | IF IDENT THEN expr
     | EDGE_EXIST_LEFT EDGE_VAR
       EDGE_EXIST_RIGHT expr
     | EDGE_ALL_LEFT EDGE_VAR
       EDGE_ALL_RIGHT expr
     ;

WHITE_SPACE ::= [ \t]+ ;
IDENT   ::= [_]* ([a-zA-Z] | '\\' [[:print:]])
            ([a-zA-Z0-9_'] | '\\' [[:print:]])* ;
NUMBER  ::= [0-9]+ ;
STRING  ::= '\"' ((([[:print:]] {-}
            [\\\"]) | '\\\\' | '\\\"')* '\"' ;
CHUNK   ::= #([0-9a-fA-F] [0-9a-fA-F])*# ;
OPERATOR  ::= [~!@<>=\-+/?&\\|*\[\]]+ ;
ENDOFLINE ::= '\n' | '\r' | '\n\r' | '\r\n' ;
LPAR      ::= '(' ;
RPAR      ::= ')' ;
DOT       ::= '.' ;
COLON     ::= ':' ;
VALUE     ::= STRING | CHUNK ;
CONSTANT    ::= IDENT ;

PREFIX1     ::= IDENT | OPERATOR ;
PREFIX2     ::= IDENT | OPERATOR ;
...
PREFIX9     ::= IDENT | OPERATOR ;

POSTFIX1    ::= IDENT | OPERATOR ;
POSTFIX2    ::= IDENT | OPERATOR ;
...
POSTFIX9    ::= IDENT | OPERATOR ;

BIN1        ::= IDENT | OPERATOR ;
BIN2        ::= IDENT | OPERATOR ;
...
BIN11       ::= IDENT | OPERATOR ;

STATE_VAR   ::= IDENT ;
EDGE_VAR    ::= IDENT ;
MU_SYM      ::= IDENT | OPERATOR ;
NU_SYM      ::= IDENT | OPERATOR ;
EXISTS_SYM  ::= IDENT | OPERATOR ;
ALL_SYM     ::= IDENT | OPERATOR ;
EDGE_EXIST_LEFT  ::= IDENT | OPERATOR ;
EDGE_EXIST_RIGHT ::= IDENT | OPERATOR ;
```

A.2 Effective grammar for CTL

```
expr ::= LPAR expr RPAR
     | STATE_VAR
     | IDENT
     | EDGE_VAR
     | VALUE
     | NUMBER
     | CONSTANT
     | expr BIN1 expr
     | expr BIN2 expr
     | expr BIN3 expr
     | expr BIN4 expr
     | expr BIN7 expr
     | expr BIN8 expr
     | expr BIN9 expr
     | expr BIN10 expr
     | expr BIN11 expr
     | PREFIX5 expr
     | PREFIX6 expr
     ;

WHITE_SPACE ::= [ \t]+ ;
IDENT   ::= [_]* ([a-zA-Z] | '\\' [[:print:]])
            ([a-zA-Z0-9_'] | '\\' [[:print:]])* ;
NUMBER  ::= [0-9]+ ;
STRING  ::= '\"' ((([[:print:]] {-}
            [\\\"]) | '\\\\' | '\\\"')* '\"' ;
CHUNK   ::= #([0-9a-fA-F] [0-9a-fA-F])*# ;
OPERATOR  ::= [~!@<>=\-+/?&\\|*\[\]]+ ;
ENDOFLINE ::= '\n' | '\r' | '\n\r' | '\r\n' ;
LPAR      ::= '(' ;
RPAR      ::= ')' ;
DOT       ::= '.' ;
COLON     ::= ':' ;
VALUE     ::= STRING | CHUNK ;

STATE_VAR ::= IDENT ;
EDGE_VAR ::= IDENT ;

CONSTANT ::= CTL_FALSE | CTL_TRUE | CTL_MAYBE ;
BIN1     ::= CTL_MULT | CTL_DIV | CTL_REM ;
BIN2     ::= CTL_ADD | CTL_SUB ;
BIN3     ::= CTL_LT | CTL_LEQ
         | CTL_GT | CTL_GEQ ;
BIN4     ::= CTL_EQ | CTL_NEQ | CTL_EN ;
PREFIX5  ::= CTL_NOT ;
PREFIX6  ::= CTL_EXIST | CTL_ALL | CTL_GLOBALLY
         | CTL_FUTURE | CTL_NEXT ;
BIN7     ::= CTL_AND ;
BIN8     ::= CTL_OR ;
BIN9     ::= CTL_EQUIV ;
BIN10    ::= CTL_IMPLY ;
BIN11    ::= CTL_UNTIL ;

CTL_FALSE      ::= 'true'
CTL_TRUE       ::= 'false'
CTL_MAYBE      ::= 'maybe'
CTL_MULT       ::= '*'
CTL_DIV        ::= '/'
CTL_REM        ::= '%'
```

```
CTL_ADD        ::= '+'
CTL_SUB        ::= '-'
CTL_LT         ::= '<'
CTL_LEQ        ::= '<='
CTL_GT         ::= '>'
CTL_GEQ        ::= '>='
CTL_EQ         ::= '=='
CTL_NEQ        ::= '!='
CTL_EN         ::= '??'
CTL_NOT        ::= '!'
CTL_EXIST      ::= 'E'
CTL_ALL        ::= 'A'
CTL_GLOBALLY   ::= '[]'
CTL_FUTURE     ::= '<>'
CTL_NEXT       ::= 'X'
CTL_AND        ::= '&&'
CTL_OR         ::= '||'
CTL_EQUIV      ::= '<->'
CTL_IMPLY      ::= '->'
CTL_UNTIL      ::= 'U'
```

A.3 Effective grammar of LTL

```
expr ::= LPAR expr RPAR
     | STATE_VAR
     | IDENT
     | EDGE_VAR
     | VALUE
     | NUMBER
     | CONSTANT
     | expr BIN1 expr
     | expr BIN2 expr
     | expr BIN3 expr
     | expr BIN4 expr
     | expr BIN7 expr
     | expr BIN8 expr
     | expr BIN9 expr
     | expr BIN10 expr
     | expr BIN11 expr
     | PREFIX5 expr
     | PREFIX6 expr
     ;

WHITE_SPACE ::= [ \t]+ ;
IDENT   ::= [_]* ([a-zA-Z] | '\\' [[:print:]])
            ([a-zA-Z0-9_'] | '\\' [[:print:]])* ;
NUMBER  ::= [0-9]+ ;
STRING  ::= '\"' (([[:print:]] {-}
            [\\\"]) | '\\\\' | '\\\"')* '\"' ;
CHUNK   ::= #([0-9a-fA-F] [0-9a-fA-F])*# ;
OPERATOR  ::= [~!@<>=\-+/?&\\|*\[\]]+ ;
ENDOFLINE ::= '\n' | '\r' | '\n\r' | '\r\n' ;
LPAR      ::= '(' ;
RPAR      ::= ')' ;
DOT       ::= '.' ;
COLON     ::= ':' ;
VALUE     ::= STRING | CHUNK ;

STATE_VAR ::= IDENT ;
EDGE_VAR ::= IDENT ;

CONSTANT ::= LTL_FALSE | LTL_TRUE | LTL_MAYBE ;
BIN1     ::= LTL_MULT | LTL_DIV | LTL_REM ;
BIN2     ::= LTL_ADD | LTL_SUB ;
BIN3     ::= LTL_LT | LTL_LEQ
         | LTL_GT | LTL_GEQ ;
BIN4     ::= LTL_EQ | LTL_NEQ | LTL_EN ;
PREFIX5  ::= LTL_NOT ;
PREFIX6  ::= LTL_GLOBALLY | LTL_FUTURE
         | LTL_NEXT ;
BIN7     ::= LTL_AND ;
BIN8     ::= LTL_OR ;
BIN9     ::= LTL_EQUIV ;
BIN10    ::= LTL_IMPLY ;
BIN11    ::= LTL_UNTIL | LTL_WEAK_UNTIL
         | LTL_RELEASE ;

LTL_FALSE      ::= 'true' ;
LTL_TRUE       ::= 'false' ;
LTL_MAYBE      ::= 'maybe' ;
LTL_MULT       ::= '*' ;
LTL_DIV        ::= '/' ;
LTL_REM        ::= '%' ;
LTL_ADD        ::= '+' ;
LTL_SUB        ::= '-' ;
LTL_LT         ::= '<' ;
LTL_LEQ        ::= '<=' ;
LTL_GT         ::= '>' ;
LTL_GEQ        ::= '>=' ;
LTL_EQ         ::= '==' ;
LTL_NEQ        ::= '!=' ;
LTL_EN         ::= '??' ;
LTL_NOT        ::= '!' ;
LTL_GLOBALLY   ::= '[]' ;
LTL_FUTURE     ::= '<>' ;
LTL_NEXT       ::= 'X' ;
LTL_AND        ::= '&&' ;
LTL_OR         ::= '||' ;
LTL_EQUIV      ::= '<->' ;
LTL_IMPLY      ::= '->' ;
LTL_UNTIL      ::= 'U' ;
LTL_WEAK_UNTIL ::= 'W' ;
LTL_RELEASE    ::= 'R' ;
```

A.4 Effective grammar for mu-calculus

```
expr ::= LPAR expr RPAR
     | STATE_VAR
     | IDENT
     | EDGE_VAR
     | VALUE
     | NUMBER
     | CONSTANT
     | expr BIN1 expr
     | expr BIN2 expr
     | expr BIN3 expr
     | expr BIN4 expr
     | expr BIN7 expr
     | expr BIN8 expr
     | PREFIX5 expr
     | MU_SYM IDENT DOT expr
     | NU_SYM IDENT DOT expr
     | EDGE_EXIST_LEFT EDGE_VAR
       EDGE_EXIST_RIGHT expr
     | EDGE_ALL_LEFT EDGE_VAR
       EDGE_ALL_RIGHT expr
     ;

WHITE_SPACE ::= [ \t]+ ;
IDENT   ::= [_]* ([a-zA-Z] | '\\' [[:print:]])
            ([a-zA-Z0-9_'] | '\\' [[:print:]])* ;
NUMBER  ::= [0-9]+ ;
STRING  ::= '\"' (([[:print:]] {-}
            [\\\"]) | '\\\\' | '\\\"')* '\"' ;
CHUNK   ::= #([0-9a-fA-F] [0-9a-fA-F])*# ;
OPERATOR  ::= [~!@<>=\-+/?&\\|*\[\]]+ ;
ENDOFLINE ::= '\n' | '\r' | '\n\r' | '\r\n' ;
LPAR      ::= '(' ;
```

```
RPAR      ::= ')' ;
DOT       ::= '.' ;
COLON     ::= ':' ;
VALUE     ::= STRING | CHUNK ;

STATE_VAR ::= IDENT ;
EDGE_VAR ::= IDENT ;

CONSTANT ::= MU_FALSE | MU_TRUE | MU_MAYBE ;
BIN1     ::= MU_MULT | MU_DIV | MU_REM ;
BIN2     ::= MU_ADD | MU_SUB ;
BIN3     ::= MU_LT | MU_LEQ | MU_GT | MU_GEQ ;
BIN4     ::= MU_EQ | MU_NEQ | MU_EN ;
PREFIX5  ::= MU_NOT ;
BIN6     ::= MU_AND ;
BIN7     ::= MU_OR ;
BIN8     ::= MU_NEXT | MU_EXIST | MU_ALL ;
MU_SYM   ::= MU_MU ;
NU_SYM   ::= MU_NU ;
EDGE_EXIST_LEFT  ::= MU_EDGE_EXIST_LEFT ;
EDGE_EXIST_RIGHT ::= MU_EDGE_EXIST_RIGHT ;
EDGE_ALL_LEFT    ::= MU_EDGE_ALL_LEFT ;
EDGE_ALL_RIGHT   ::= MU_EDGE_ALL_RIGHT ;

MU_FALSE ::= 'true'
MU_TRUE  ::= 'false'
MU_MAYBE ::= 'maybe'
MU_MULT  ::= '*'
MU_DIV   ::= '/'
MU_REM   ::= '%'
MU_ADD   ::= '+'
MU_SUB   ::= '-'
MU_LT    ::= '<'
MU_LEQ   ::= '<='
MU_GT    ::= '>'
MU_GEQ   ::= '>='
MU_EQ    ::= '=='
MU_NEQ   ::= '!='
MU_EN    ::= '??'
MU_NOT   ::= '!'
MU_AND   ::= '&&'
MU_OR    ::= '||'
MU_NEXT  ::= 'X'
MU_EXIST ::= 'E'
MU_ALL   ::= 'A'
MU_MU    ::= 'mu'
MU_NU    ::= 'nu'
MU_EDGE_EXIST_LEFT  ::= '<'
MU_EDGE_EXIST_RIGHT ::= '>'
MU_EDGE_ALL_LEFT    ::= '['
MU_EDGE_ALL_RIGHT   ::= ']'
```

B. MODULAR GRAMMAR CONVERTER GRAMMAR

```
grammar ModGram;

gram : module+ ;

module : 'module' LC_NAME ';'
         ('using' LC_NAME ';')* (gram_rule ';')*;

gram_rule : left_hand_side '::=' right_hand_side
          | left_hand_side '<-' LC_NAME
            '.' left_hand_side
          | left_hand_side '<=' LC_NAME
            '.' left_hand_side
          | left_hand_side '<=*' LC_NAME
            '.' left_hand_side
          | left_hand_side ':/=' right_hand_side
          ;

left_hand_side : LC_NAME | UC_NAME ;
right_hand_side : regexp+ ;
regexp : LC_NAME
       | UC_NAME
       | LC_NAME '.' LC_NAME
       | LC_NAME '.' UC_NAME
       | '$'
       | STRING
       | CHARS
       | '(' regexp+ ')'
       | '+'
       | '*'
       | '?'
       | '~'
       | '|'
       ;

LC_NAME: [a-z] [a-z_0-9]*;
UC_NAME: [A-Z] [A-Z_0-9]*;
STRING: '\'' ('\\'. | ~('\''|'\\'))* '\'';
CHARS: '[' (~('\\'| ']' | '[') | '\\'.)* ']';
WS: [ \t\n\r]+ -> skip;
```

C. LTSMIN MODULAR GRAMMAR 1

```
module main ;
using ctl ;
using ltl ;
using mucalc ;
using pred ;

expr <= ctl.expr ;
expr <= ltl.expr ;
expr <= mucalc.expr ;
expr <= pred.expr ;
```

```
module mucalc ;
using pred ;

expr ::= MU pred.VAR DOT expr
     | NU pred.VAR DOT expr
     | EDGE_EXIST_LEFT pred.VAR
       EDGE_EXIST_RIGHT expr
     | EDGE_ALL_LEFT pred.VAR
       EDGE_ALL_RIGHT expr
     | pred.NEXT expr
     | pred.EXIST expr
     | pred.ALL expr
     ;

MU ::= 'mu' ;
NU ::= 'nu' ;
EDGE_EXIST_LEFT  ::= '<' ;
EDGE_EXIST_RIGHT ::= '>' ;
EDGE_ALL_LEFT    ::= '[' ;
EDGE_ALL_RIGHT   ::= ']' ;
DOT   ::= '.' ;
```

```
module ctl ;
using pred;

expr ::= pred.EXIST expr
     | pred.ALL expr
     | GLOBALLY expr
     | FUTURE expr
     | pred.NEXT expr
     | expr pred.UNTIL expr
     ;

GLOBALLY ::= 'G' ;
FUTURE ::= 'F' ;
```

```
module ltl ;
using pred ;

expr ::= GLOBALLY expr
     | FUTURE expr
     | pred.NEXT expr
     | expr pred.UNTIL expr
     | expr WEAK_UNTIL expr
     | expr RELEASE expr
     ;

GLOBALLY   ::= '[]' ;
FUTURE     ::= '<>' ;
WEAK_UNTIL ::= 'W' ;
RELEASE    ::= 'R' ;
```

```
module pred ;

expr ::= VAR
     | NUMBER
     | TRUE
     | FALSE
     | MAYBE
     | STRING
     | CHUNK
     | LPAR expr RPAR
     | expr MULT expr
     | expr DIV expr
     | expr REM expr
     | expr ADD expr
     | expr SUB expr
     | expr LT expr
     | expr LEQ expr
     | expr GT expr
     | expr GEQ expr
     | expr EN expr
     | expr EQ expr
     | expr NEQ expr
     | expr AND expr
     | expr OR expr
     | NOT expr
     | expr EQUIV expr
     | expr IMPLY expr
     ;

STRING    ::= '"' (~[\\"] | '\\'$)* '"' ;
CHUNK     ::= '#' ([0-9a-fA-F]
              [0-9a-fA-F])* '#' ;
NUMBER    ::= [0-9]+ ;
TRUE      ::= 'true' ;
FALSE     ::= 'false' ;
MAYBE     ::= 'maybe' ;
VAR       ::= '_'* ([a-zA-Z] | '\\'$)
              ([a-zA-Z0-9_'] | '\\'$)* ;

LPAR  ::= '(' ;
RPAR  ::= ')' ;
MULT  ::= '*' ;
DIV   ::= '/' ;
REM   ::= '%' ;
ADD   ::= '+' ;
SUB   ::= '-' ;
LT    ::= '<' ;
LEQ   ::= '<=' ;
GT    ::= '>' ;
GEQ   ::= '>=' ;
EQ    ::= '==' ;
NEQ   ::= '!=' ;
EN    ::= '??' ;
NOT   ::= '!' ;
AND   ::= '&&' ;
OR    ::= '||' ;
EQUIV ::= '<->' ;
IMPLY ::= '->' ;

NEXT  ::= 'X' ;
EXIST ::= 'E' ;
ALL   ::= 'A' ;
UNTIL ::= 'U' ;
```

D. LTSMIN MODULAR GRAMMAR 2

```
module main ;
using ctl ;
using ltl ;
using mucalc ;

expr ::= ctl.expr
     | ltl.expr
     | mucalc.expr
     ;
```

```
module mucalc ;
using pred ;
using bool ;

expr <= bool.expr ;
expr ::= MU pred.VAR DOT expr
     | NU pred.VAR DOT expr
     | EDGE_EXIST_LEFT pred.VAR
       EDGE_EXIST_RIGHT expr
     | EDGE_ALL_LEFT pred.VAR
       EDGE_ALL_RIGHT expr
     | pred.NEXT expr
     | pred.EXIST expr
     | pred.ALL expr
     | pred.comp_expr
     ;

MU ::= 'mu' ;
NU ::= 'nu' ;
EDGE_EXIST_LEFT  ::= '<' ;
EDGE_EXIST_RIGHT ::= '>' ;
EDGE_ALL_LEFT    ::= '[' ;
EDGE_ALL_RIGHT   ::= ']' ;
DOT   ::= '.' ;
```

```
module ctl ;
using bool ;
using pred ;

expr <= bool.expr ;
expr <= bool.expr_extra ;
expr ::= pred.EXIST expr
     | pred.ALL expr
     | pred.GLOBALLY expr
     | pred.FUTURE expr
     | pred.NEXT expr
     | expr pred.UNTIL expr
     | pred.comp_expr
     ;
```

```
module ltl ;
using pred ;
using bool ;

expr <= bool.expr ;
expr <= bool.expr_extra ;
expr ::= pred.GLOBALLY expr
     | pred.FUTURE expr
     | pred.NEXT expr
     | expr pred.UNTIL expr
     | expr WEAK_UNTIL expr
     | expr RELEASE expr
     | pred.comp_expr
     ;

WEAK_UNTIL ::= 'W' ;
RELEASE    ::= 'R' ;
```

```
module pred ;
using bool ;

comp_expr ::= calc_expr EQ calc_expr
            | calc_expr NEQ calc_expr
            | calc_expr LT calc_expr
            | calc_expr LEQ calc_expr
            | calc_expr GT calc_expr
            | calc_expr GEQ calc_expr
            | calc_expr EN calc_expr
            ;

calc_expr ::= VAR
            | NUMBER
            | bool.LPAR calc_expr bool.RPAR
            | calc_expr MULT calc_expr
            | calc_expr DIV calc_expr
            | calc_expr REM calc_expr
            | calc_expr ADD calc_expr
            | calc_expr SUB calc_expr
            ;

STRING    ::= '"' (~[\\"] | '\\'$)* '"' ;
CHUNK     ::= '#' ([0-9a-fA-F]
              [0-9a-fA-F])* '#' ;
NUMBER    ::= [0-9]+ ;
VAR       ::= '_'* ([a-zA-Z] | '\\'$)
              ([a-zA-Z0-9_'] | '\\'$)* ;

MULT  ::= '*' ;
DIV   ::= '/' ;
REM   ::= '%' ;
ADD   ::= '+' ;
SUB   ::= '-' ;
LT    ::= '<' ;
LEQ   ::= '<=' ;
GT    ::= '>' ;
GEQ   ::= '>=' ;
EQ    ::= '==' ;
NEQ   ::= '!=' ;
EN    ::= '??' ;

GLOBALLY ::= '[]' ;
FUTURE   ::= '<>' ;
NEXT     ::= 'X' ;
EXIST    ::= 'E' ;
ALL      ::= 'A' ;
UNTIL    ::= 'U' ;
```

```
module bool;

expr ::= expr AND expr
     | expr OR expr
     | NOT expr
     | TRUE
     | FALSE
     | MAYBE
     | LPAR expr RPAR
     ;

expr_extra ::= expr_extra EQUIV expr_extra
           | expr_extra IMPLY expr_extra
           ;

LPAR  ::= '(' ;
RPAR  ::= ')' ;
TRUE  ::= 'true' ;
FALSE ::= 'false' ;
MAYBE ::= 'maybe' ;
NOT   ::= '!' ;
AND   ::= '&&' ;
OR    ::= '||' ;
EQUIV ::= '<->' ;
IMPLY ::= '->' ;
```