Compositional Maude: syntax and usage

Ó. Martín, A. Verdejo, N. Martí-Oliet
Facultad de Informática
Universidad Complutense de Madrid, Spain
{omartins,jalberto,narciso}@ucm.es

December 21, 2022

1 Introduction

This document describes the usage of a piece of software. It is related to our work on bringing compositional specification and verification to rewriting logic.

We have proposed an extension to the language Maude to allow for compositional specification and verification; we refer to it as *compositional Maude*. Our tool accepts specifications written in this extended language and performs a translation into standard Maude. This document describes the syntax that we have proposed and that our tool accepts. The tool is functional, but must be considered a work in progress. Indeed, at present, it does not accept all standard Maude syntax. We describe the usage of the tool and the points where we fall short.

This document is mainly about using the tool. Explanations on its internal workings and the theoretical basis behind it is to be found in the papers mentioned on our webpage `http://maude.ucm.es/syncprod/`.

2 The syntax of compositional Maude

In addition to the standard modules in Maude, our extension introduces egalitarian modules. These can be atomic or nonatomic. We describe their syntax in the following two sections. Then we describe the syntax for assignment synchronization criteria and assume/guarantee statements.

2.1 Atomic egalitarian modules

Atomic egalitarian modules are delimited by the keywords `aemod` and `endaem`. They are system modules (as opposed to functional ones). The main difference with respect to standard system modules is the use of egalitarian rewrite rules:

```
rl s =[ t ]=> s' .
```

where `s` and `s'` are terms of sort `State` and `t` is a term of sort `Trans`. These two sorts, `State` and `Trans`, with a supersort of both named `Stage`, are declared in a predefined module called `STAGE`, which can be imported into any system module. Rules can be conditional:

```
crl s =[ t ]=> s' if C .
```

We require that the module is topmost, that is, rewrites can only happen at the top of the initial term and whatever subsequent stage terms. Free variables are allowed to appear in `t`, `s'`, and `C`, although this can prevent the resulting module from being executable and model-checkable.

The last ingredient of atomic modules is the infrastructure to declare and define properties. We introduce the keyword `ppt` and a new declaration sentence marked by it:

```
ppt p : s1 ... sn -> s .
```

where `p` is an identifier for the property, and `s`, `si` are sorts. Often, `n` is 0:

```
ppt p : -> s .
```

The symbol `@` represents the evaluation of a property at a stage. The definition of the values of properties is specified by equations with the form

```
eq p @ g = ...
```

for a property `p` (maybe with parameters) and a stage term `g`. Conditional equations can certainly be used.

For an example, this is a module modeling the tick of a clock, with a property to inform about its ticking:

```
aemod CLOCK is
    ex STAGE .
    ops before after : -> State .
    ops ticking gettingReady : -> Trans .
    eq init = before .
    rl before =[ ticking ]=> after .
    rl after =[ gettingReady ]=> before .
    ppt isTicking : -> Bool .
    eq isTicking @ ticking = true .
    eq isTicking @ G:Stage = false [owise] .
endaem
```

For convenience, the constant `init` is also declared in `STAGE`, to represent the initial stage of the system.

2.2 Nonatomic egalitarian modules

The keywords `emod` and `endem` delimit nonatomic egalitarian modules. These are system modules which specify how a new system is built from existing components. The components can be atomic or nonatomic. We also allow standard Maude system modules (`mod` ... `endm`) as components, as long as they declare the needed properties.

The instruction to build a composed system has this syntax:

```
sync $M_1$ || ... || $M_n$
    on $M_i$$$p_i$ = $M_j$$$p_j$
    /\ ...
    /\ $M_k$$$p_k$ = $M_l$$$p_l$ .
```

Each M_h is a module expression, and each p_h is a property declared and defined in M_h. All subscripts range over $1, \dots, n$, and they can be repeated, that is, the same module can be used in several criteria, with the same property or a different one. The semantics of this instruction is that the systems $M_1, \dots, M_n$ can proceed in whichever way that makes the properties satisfy all the equalities at all times.

The syntax $M\$p$, with the dollar sign, is copied from the one to access elements of (an instantiation of) a theory in Maude. Indeed, the operands for the `sync`...`on` instruction may be modules received as arguments, but they may alternatively be modules previously introduced. That is, both of these structures are allowed:

```
emod M1 is ... ppt p1 : ... endem

aemod M2 is ... ppt p2 : ... endaem

emod M is
    sync M1 || M2
        on M1$p1 = M2$p2 .
endem
```

or

```
th T1 is ... ppt p1 : ... endth

th T2 is ... ppt p2 : ... endth

emod M{X1 :: T1, X2 :: T2} is
    sync X1 || X2
        on X1$p1 = X2$p2 .
endem
```

An `emod` must contain exactly one `sync ...on` instruction and zero or more property declarations and `inh` statements. An `inh` statement (short for *inherit*) specifies which properties from the components are inherited by the new module, and with what names. Only properties that are explicitly inherited in this way count as properties of the composed module. Thus, the general shape of an `inh` statement is

```
inh $p'$ = $M$$$p$ .
```

specifying that the property p from module M is inherited in the current module with name p'. Properties defined by an `inh` statement must have been previously declared.

The following is an example `emod`. It assumes the modules `CLOCK` and `CLOCK'` define each a Boolean property `isTicking`. It specifies that both tick at the same time. In turn, it defines the property `isFirstTicking`.

```
emod SYNCED-CLOCKS is
```

```
    sync CLOCK || CLOCK'
        on CLOCK$isTicking = CLOCK'$isTicking .
    ppt isFirstTicking : -> Bool .
    inh isFirstTicking = CLOCK$isTicking .
    endem
```

2.3 Assignment synchronization criteria

In addition to the equality synchronization criteria presented above, we allow *assignment synchronization criteria*, replacing `=` by `:=`. That is, both of these

$M_1\$p_1$ `=` $M_2\$p_2$

$M_1\$p_1$ `:=` $M_2\$p_2$

are allowed as synchronization criteria in a `sync`...`on` instruction. The semantics is the same but the use of `:=` conveys the idea that the value on which the properties agree is chosen by M_2, while M_1 is ready to go on with any value it is handed. That is, it emulates value passing.

The advantage of using assignment criteria, as well as the requirements implied by its use, will be made clear by a simple example. Consider a sender/receiver system. The module `RECEIVER` includes the rule

```
rl readyToReceive =[ receiving ]=> received X .
```

for a variable `X`, and it defines a property `valueReceived` by

```
eq valueReceived @ received X = X .
```

That rule is not executable by standard means, because it has a free variable on the right-hand side.

Meanwhile, the module `SENDER` includes the rule

```
rl readyToSend Y =[ sending Y ]=> sent Y .
```

and this definition for the property `valueSent`

```
eq valueSent @ sent Y = Y .
```

We want to compose both systems so that the value one sends is the value the other receives. We use an assignment criterion:

```
sync SENDER || RECEIVER
    on RECEIVER$valueReceived := SENDER$valueSent .
```

Our tool, when fed with those modules, composes the individual rules, producing in first instance the composed rule

```
crl < receiving, sending Y > => < received X, sent Y >
    if valueReceived @ received X := valueSent @ sent Y .
```

The synchronization criterion has been translated into a condition on the destination state. As it stands, this is not a valid matching condition in Maude (its left-hand side is not what is called a *pattern* in [@ClavelDEELMMRT2020MaudeM31]), but both sides are statically evaluated, resulting in

```
crl < receiving, sending Y > => < received X, sent Y >
    if X := Y .
```

This is executable, but it would have not been executable with an equational condition: `if X = Y`. This is the advantage of assignment criteria: they allow producing executable composed modules from nonexecutable components.

The fact that the split translates assignment criteria into matching conditions, thus, puts an extra requirement on the definition of the properties involved. Namely, the expression to the left of the `:=` sign must be statically equationally reducible to a pattern. In most cases in which we use `:=` in our specifications, the expression to the left reduces to a variable, which is a simple kind of pattern.

2.4 Assume/guarantee statements

To be ready for assume/guarantee-style compositional verification, we introduce a new kind of statement:

```
ag $\alpha$ |> $\gamma$ .
```

where $\alpha$ is the temporal formula representing the assumption, $\gamma$ is the one representing the guarantee, `ag` is a new keyword, and the symbol `|>` is the ASCII rendering for $\rhd$. Thus, such a statement appearing in a module `M` represents the assertion $\mathtt{M} \models \alpha \rhd \gamma$.

Such statements are allowed in any system module, either `aemod`, `emod`, or even `mod`. The constant formula `True` can be used to the left of `|>` when no assumption is needed. Several `ag` statements are allowed in the same module, with different assumptions or guarantees. They are also allowed in theories, in which case they are to be understood as expressing requirements for each instance to satisfy. But if we want to state that a particular instantiation of the theory does satisfy the `ag` statement, we have to repeat it in the body of the instantiation.

Both formulas, $\alpha$ and $\gamma$, have to be elements of the sort `Formula` as declared in the standard Maude module `LTL`. However, we depart here from the standard in that we use properties in formulas, instead of the usual atomic propositions. That is, we assume the declaration `subsort Bool < Formula .` instead of `subsort Prop < Formula .` Only Boolean properties are allowed in formulas, which are in essence the same thing as the usual atomic propositions. Still, given that all our developments are based on properties, we feel that using them to the very end is the right thing to do.

To the module `SYNCED-CLOCKS` above, for example, the following can be added:

```
ag True |> [] <> isFirstTicking .
```

3 Usage

The implementation is to be fed with a specification written in Compositional Maude, and it produces an equivalent standard Maude specification, that can then be run, analyzed and verified.

To run the code, a decently recent version of Python 3 is needed. Also needed are the packages Lark (https://github.com/lark-parser/lark) for EBNF parsing, and `maude-bindings` (https://github.com/fadoss/maude-bindings), for calling Maude's engine from Python.

The command line to execute is

```
python cmaude c|v <infile> <module>
```

where:

- a `c` or a `v` must be specified, to obtain one of the two possible translations (explained below);
- `<infile>` is the name of the file with the compositional specification, usually with `.cmaude` extension;
- `<module>` is the name of the module in the `<infile>` that represents the whole composed system, and that we are interested in analyzing.

Also, `cmaude` is the name of the directory I have placed my Python code into. It can be named otherwise (and the command line changed accordingly). Python, when given the name of a directory, looks inside for a file named `__main__.py` and executes it.

If the input file is named `whatever.cmaude`, the output is named `whatever-c.maude` or `whatever-v.maude`, according to the translation we chose in the command line.

The output file contains the translation of `<module>` and the translations of any other modules that `<module>` depends on.

We want to make it clear that, at present, our tool only performs the translation, and hands the standard-Maude result to the user, who can then use existing Maude tools on it.

4 The two translations

As observed above, two different translations are provided by our tool, corresponding to choosing `c` (for *compose*) or `v` (for *verify*) in the command line. Strictly speaking, only the first one is an actual translation: the second produces only the needed machinery to verify the global system. Let us be a little more clear.

The `c` option outputs, as promised, a standard Maude module that is equivalent to the compositionally specified original one. Usually, some auxiliary modules need to be translated and output as well. Loosely speaking, the modules

```
fmod M1 is ... endfm
emod M2 is ... endem
aemod M3 is ... endaem
emod M is
    pr M1 .
    sync M2 || M3 on ...
    ...
endem
```

are translated into

```
fmod M1 is ... endfm
mod M is
    pr M1 .
    ⟨statements for the composition of M2 and M3⟩
    ...
endm
```

The second translation is useful when we are aiming only at verification, not execution, and each component includes its own `ag` statements. The schematic example below may help clarify all the wording that follows.

In this case, we interpret that the specifier is stating that

- each component satisfies the `ag` statements included in it, and
- the global `ag` statements, the ones included in the composed system, follow from the ones in the components.

In this case, we do not need to perform a complete translation, we only have to check that the global `ag` statements follow from the components'.

What our implementation does, then, is:

- it recursively deals with each component and its `ag` statements until reaching the base cases (see next), and
- produces a functional (not system) module containing the formula that represents the fact that the `ag` statements from the components imply the ones in the composed module.

The base cases for this procedure are:

- `aemod`s,
- standard `mod`s,
- `emod`s some of whose components do not contain `ag` statements.

It is then left to the user the task of verifying the translated base cases (probably with the use of Maude's model checker), and proving the `ag`'s from the components imply the one in the composed system (probably with the use of Maude's tautology checker).

Each `ag` is translated into a constant operator of sort `Formula`, with names `<ag0>`, `<ag1>`... But in an `emod` all whose components have `ag`s, each `ag` gives also rise to new formula with names `<ded0>`, `<ded1>`... each stating that the components `ag`'s imply the corresponding global `ag`.

For a very schematic example, consider this compositional specification:

```
aemod M1 is
    ...
    ag a1 |> g1 .
endaem

aemod M2 is
```

```
    ...
    --- no ag
endaem

emod M3 is
    sync M1 || M2 on ...
    ag a3 |> g3 .
endem

aemod M4 is
    ...
    ag a4 |> g4 .
endaem

emod M5 is
    sync M3 || M4 on ...
    ag a5 |> g5 .
    ag a5' |> g5' .
endem
```

When processed by our tool with the `v` option, it would produce:

```
mod M3 is
    ... --- statements for the composition of M1 and M2
    eq <ag0> = a3 -> g3 .
endm

mod M4 is
    ...
    eq <ag0> = a4 -> g4 .
endm

fmod M5 is
    ...
    eq <M3|<ag0>> = <M3|a3> -> <M3|g3> .
    eq <M4|<ag0>> = <M4|a4> -> <M4|g4> .
    eq <ag0> = a5 -> g5 .
    eq <ded0> = (<M3|<ag0>> /\ <M4|<ag0>>) -> <ag0> .
    eq <ag1> = a5' -> g5' .
    eq <ded1> = (<M3|<ag0>> /\ <M4|<ag0>>) -> <ag1> .
endfm
```

The points to note are:

- The verification of the `ag` statement in module `M3` cannot be made compositionally, because one of the components, `M2` does not include an `ag` statement of its own. Thus, `M3` is processed as if with the `c` option, and neither `M1`, nor `M2` are output.
- The formulas named `<ag0>` in `M3` and `M4` are direct translations of the corresponding `ag` statements. The user should model check each of these modules against its `<ag0>`.

- The global module `M5` is transformed into a functional module. The result does not include the statements for the composition of `M3` and `M4`, but just definitions of formulas. Each `ag` statement in `M5` is, first, literally translated into a formula `<agN>` and, second, produces a `<dedN>` formula stating that the component's `ag` statements imply each `<agN>`.

- The operators `<ag0>` from the components are qualified with their component-module name when inserted into `M5`.

- A complete verification of `M5` consists in verifying `M3` and `M4` satisfy their `<ag0>` (maybe by using Maude's model checker), and checking that `<ded0>` and `<ded1>` indeed hold (maybe by using Maude's tautology checker).

5 Important fine print to have in mind

5.1 On the qualification of names

When we translate a composed module (that is, an `emod`) into a standard one (a `mod`), it may happen that names from different component modules clash. That is, that operators or sorts from different components are called the same just by chance. We want to avoid this. The way Maude deals with this when importing modules into others is to trust the coder, assume no clash will happen, but warn the user when they do happen. But we insist all the time in the importance of considering each component as an individual system, coded with no regard to other possible components with which it may be composed later. Thus, we think it is necessary that our tool deals properly with that. The solution is that we qualify names with the name of the module they come from, so that `SortName` in module `ModName`, when inserted in the translation of composed system, gets renamed as `<ModName|SortName>`. Of course, the particular pattern used for the renaming is of no importance (it is isolated in a function in the code), as long as there is a minimal chance that clashes still occur. However, this is more easily said than done. We discuss next the difficulties with this approach, how we have solved them, and with which limitations.

We qualify names of sorts, operators, and variables, both in the text of the module and in the modules imported in `including` mode if they are functional modules or imported in any way if they are system modules. (See the first author's PhD thesis, *Composition in rewriting logic*, for the rationale for this.) Renaming them in their declarations is an easy task, as they are easy to spot. Quite more difficult is to rename them when they occur inside a term. Our grammar definition and our parser are not able to parse inside terms. That is, they are able to delimit the lhs of an equation, but not inside it. The reason is that it depends not on the grammar of Maude (either extended for compositionality or not) but in the signature declared in the very Maude module. Maude (and Full Maude) performs a two-pass parsing, the first pass being like ours, the second one being used to solve the so-called *bubbles*, that is, to parse inside terms by using the signature, which is now known thanks to the first pass. We can do this as well, because the `metaParse` function is available to us through Rubén Rubio's package `maude-bindings`. But, as we explain next, this only solves part of the problem.

So, say we have the term in its `bubble` form (that is, as a string of characters) and parsed into an object of the class `Term` as returned by Rubén's tool. Now, we need to traverse the term tree, qualifying each identifier that needs to while traversing, build a new term with the qualified pieces, and print the resulting term to a string. Traversing the term is possible, again, by using Rubén's tool. Prettyprinting is also possible. The problem is building the new term, because there are no functions for it. Also the question of which module would the produced term be a term in needs consideration: the answer is "in the renamed signature of the original module". But, after all, this renamed signature is part of the new module, the result of the translation of the composed module, so we can just take the signature of this resulting module. It remains, however, the problem with building a term. I guess Rubén could enrich his tool with functions to this aim but, at present, they do not exist.

Finally, here is our solution. We delimit the elements in the term by looking for spaces and breaking characters (in Maude they are `()[]{},`). In this easy way we can find the names that need to be qualified, perform a substring replace, and done. This, indeed, works well for operators declared in the standard prefix way, that is, with no underscores used as placeholders. Underscores make everything more complex. We proceed by considering each part of the name as a name to qualify in itself, but this is error-prone. For example, the same piece can be part of more that one identifier and, in this case, we qualify it everywhere we find it. Or, for another example, a comma (`,`) can be part of such an identifier and, in this case, we will qualify all commas, even the ones that are part of Maude syntax, like for separating arguments in a function call. Also, empty syntax is not renamed at all.

By the way, the names to be qualified are also split by backquotes that represent blank spaces, that is, the ones not followed by a breaking character. This is also error-prone!

5.2 Other fine print

- Our implementation accepts rules with free variables on the rhs. When split (that is, translated), these produce rules that are not valid for Maude's engine. However, when a module including such a rule is composed with another with an appropriate assignment criterion, the variable stops being free in the resulting composed rule. Therefore, free variables must be used with care.

- To each module that includes an `ag` statement, the translation adds the importation `ex MODEL-CHECKER * (sort State to <MC|State>) .`, so that, in particular, formulas can be written in the `ag`. The renaming of `State` is needed because we use the sort `State` for our own aims, and we want to avoid clashes (namely, our `State` is a subsort of `Stage`, that is in turn made a synonym of `<MC|State>`).

- Dealing with formulas (that is, terms of sort `Formula`) in the `ag` statement is made complicated by two factors. One is that it is convenient to be able to parse a formula, so as to transform it as needed. The second, more important, is that the two arguments of an `ag _ |> _ .` statement have to be formulas. Thus, it seems necessary that `Formula` be a grammatical entity. The alternative is that the parser accepts any terms there, and only afterwards it is checked (or assumed) their sort is `Formula`. At present, we have chosen the option to make it part of our extended grammar. However, when all is translated into standard Maude, the formulas are terms of sort `Formula`, as usual. Doing it like I am doing it at present has the problem that formulas have a predefined syntax, and I cannot define my own formula operators, for instance, nor auxiliary formulas (convenient in case they get too large).

- For reasons explained somewhere else (see the first author's PhD thesis), rules have to be written and properties have to be defined in such a way that there is always an equation that matches the state terms in the rule, without over-relying on the `owise` attribute. For example,

  ```
  var R : Pair .
  rl m1 | R => m2 | R .
  ...
  eq P @ (m2 | (X, Y)) = Y .
  eq P @ S:State = 0 [owise] .
  ```

  is problematic, because `m2 | R`, considering `R` as if it was a new constant, does not match the first equation and erroneously falls into the second. Instead, we would write the rule as:

  ```
  rl m1 | (X, Y) => m2 | (X, Y) .
  ```

  Or, alternatively, the equations as:

  ```
  eq P @ (m2 | R) = second(R) .
  eq P @ S:State = 0 [owise] .
  ```

  Better yet, we could entirely avoid the use of `owise` in the equations defining properties.

- Our implementation assumes that all properties are totally defined.

- Operators declared with underscores are not accepted.

- When the option `v` is used, synchronization criteria are translated by our tool into equations. Therefore, they must be written with care. For example, this

  ```
  sync A || B || C
      on A$P = B$Q
      /\ A$P = C$R .
  ```

would be translated into

```
eq A$P = B$Q .
eq A$P = C$R .
```

which is non-confluent. Instead, it must be written as

```
sync A || B || C
    on B$Q = A$P
    /\ C$R = A$P .
```

- Rules do not are allowed to have labels, even standard rules.
- Import instructions must be for only one module each. That is, no `pr M1 M2 .`, but `pr M1 . pr M2 .`
- Operators to the kind must be declared by using the `~>`. The notation `[Kind]` is not accepted.
- The option `v` outputs the translation of all modules that include one or more `ag` statements (and the ones imported by these). Some of them may be unsuitable to be fed into Maude because of fresh variables on the rhs of rules.
- In the papers and the PhD thesis, we are using `rl` and `crl` to introduce egalitarian rules, the same as for standard rules. However, our implementation, at present, needs that egalitarian rules are introduced by the keywords `erl` and `cerl`.