Universidade Federal do Rio Grande do Sul

Verites

Verification, Validation and Test Group

VERIGRAPH

Software specification and verification tool based on graph rewriting

Version 1.1.0

February 23, 2017

Andrei Costa
Guilherme Grochau Azzi
Jonas Santos Bezerra
Leila Ribeiro
Leonardo Marques Rodrigues
Rodrigo Machado

Verigraph Tutorial

February 23, 2017

Contents

1 Installation

1.1 Requirements

- GHC (The Glasgow Haskell Compiler) version 7.10 or superior
- AGG (The Attributed Graph Grammar System) version 2.1: Download at `http://www.user.tu-berlin.de/o.runge/agg/`
- Cabal 1.24 or Stack 1.1.2
- Git Version Control System (Optional)

1.2 On Linux/Mac

1.2.1 Downloading

Download the source code using git and checkout the latest stable version:

```
$ git clone https://github.com/Verites/verigraph.git
$ cd verigraph
$ git checkout 1.1.0
```

Or download the `zip` or `tar.gz` file at of the version 1.1.0 at `https://github.com/Verites/verigraph/releases`, extract the file and go the target directory.

1.2.2 Installing

If you are using Stack:

```
$ stack install
$ PATH=${PATH}:~/.local/bin
```

If you are using Cabal:

```
$ cabal install
$ PATH=${PATH}:~/.cabal/bin
```

You should also set the PATH variable in your `.bashrc` file, ensuring that the verigraph executables will be found in future sessions, after the previous steps, this can be done by this way:

```
$ echo "export PATH=${PATH}" >> ~/.bashrc
$ source ~/.bashrc
```

1.3 On Windows

1.3.1 Downloading

Download the source code using git and checkout the latest stable version:

```
$ git clone https://github.com/Verites/verigraph.git
$ cd verigraph
$ git checkout 1.1.0
```

Or download the `zip` or `tar.gz` file at of the version 1.1.0 at `https://github.com/Verites/verigraph/releases`, extract the file and go the target directory via the *command prompt*.

1.4 Installation

Download the latest Stack[1] from `https://docs.haskellstack.org/en/stable/install_and_upgrade/#windows` and follow the default installation instructions.

In the prompt command, run:

```
..\verigraph-1.1.0> stack setup
..\verigraph-1.1.0> stack install
```

[1]Installation via Cabal was not tested on Windows

2 Global Configuration

The rewriting semantics of DPO may restrict matches to monomorphisms (i.e. injective morphisms) or allow arbitrary morphisms. By default, *verigraph* will restrict matches to monomorphisms, but it will allow arbitrary matches if given the `--all-matches` flag.

Two strategies for satisfability of Negative Application Conditions (NACs) are also provided. By default, only monomorphisms between the NAC object and the instance graph are considered. Alternatively, using the flag `--partial-injective-nacs`, *verigraph* behaves like AGG and admits the collapsing of elements, as long as they are also collapsed by the match.

Presently, *AGG* is used for creating and reading graph grammar instances and analysis results of *verigraph*, by means of its `.ggx` and `.cpx` files. However, *verigraph* does not support some of AGG's features yet, such as:

- Edge types with same name but different source and/or target types.
- Inheritance
- Multiplicity
- Attributes

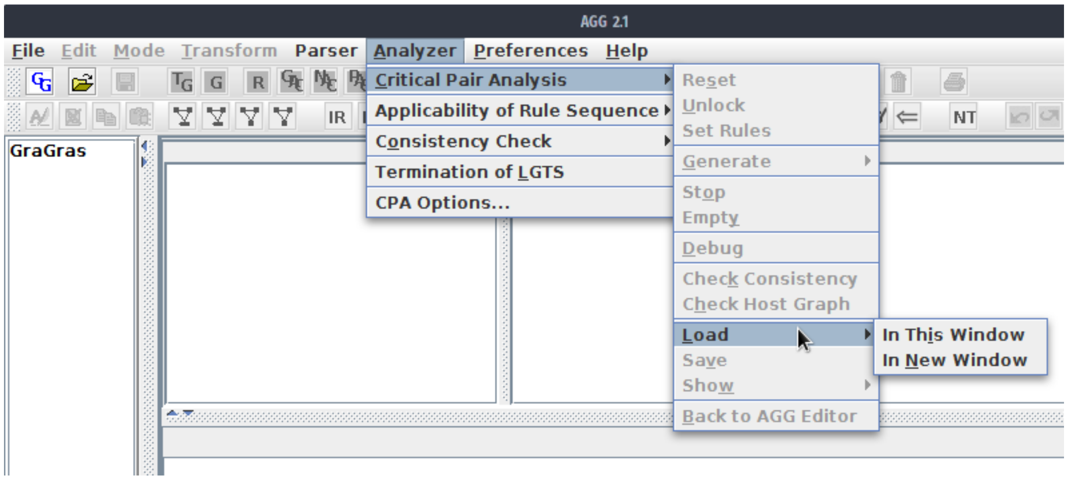
3 Critical Pair Analysis

In this section we'll analyze the conflicts and dependencies of the grammar located on `grammars/Server/server.ggx`, which models the communication between a server and a client. You may want to open it with AGG and familiarize yourself with it.

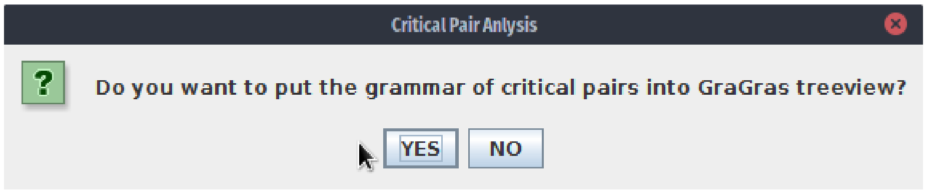
In order to obtain the conflicts and dependencies of the grammar, run:

```
$ verigraph analysis -o output.cpx grammars/Server/server.ggx
```

Then open `output.cpx` with AGG by clicking on one of the options under Analyzer > Critical Pair Analysis > Load.

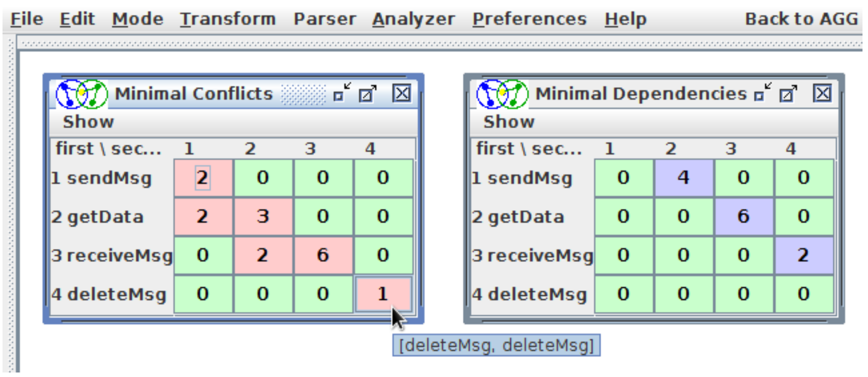


When asked about adding the grammar to the tree view, click yes. Otherwise, AGG will not display the individual conflicts in detail.



Do you want to put the grammar of critical pairs into GraGras treeview?

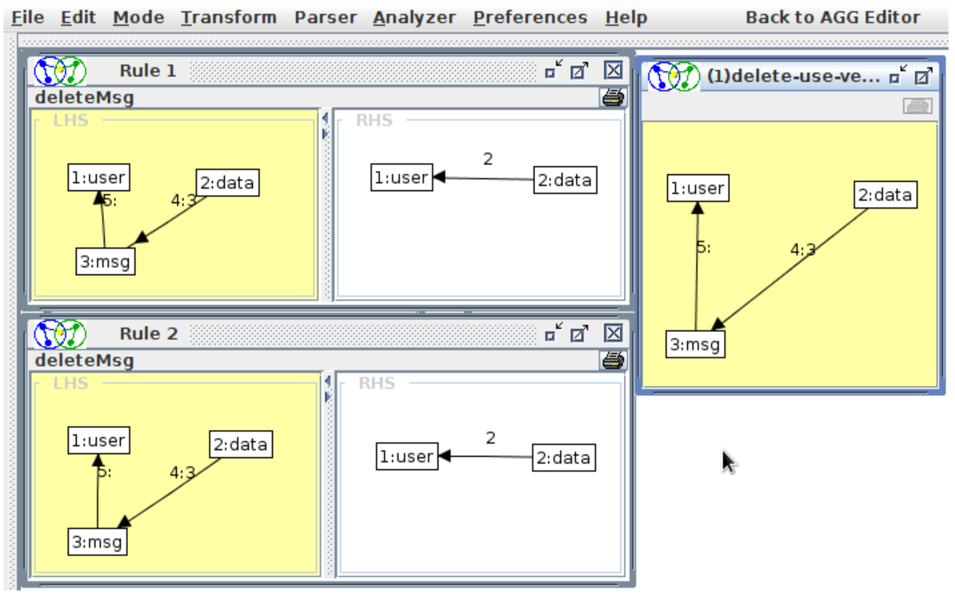
AGG will display matrices indicating the number of conflicts and dependencies between all pairs of rules. In order to visualize the individual conflicts in detail, click on a cell, for example $(4, 4)$ in the conflict matrix.



first \ sec...	1	2	3	4
1 sendMsg	2	0	0	0
2 getData	2	3	0	0
3 receiveMsg	0	2	6	0
4 deleteMsg	0	0	0	1

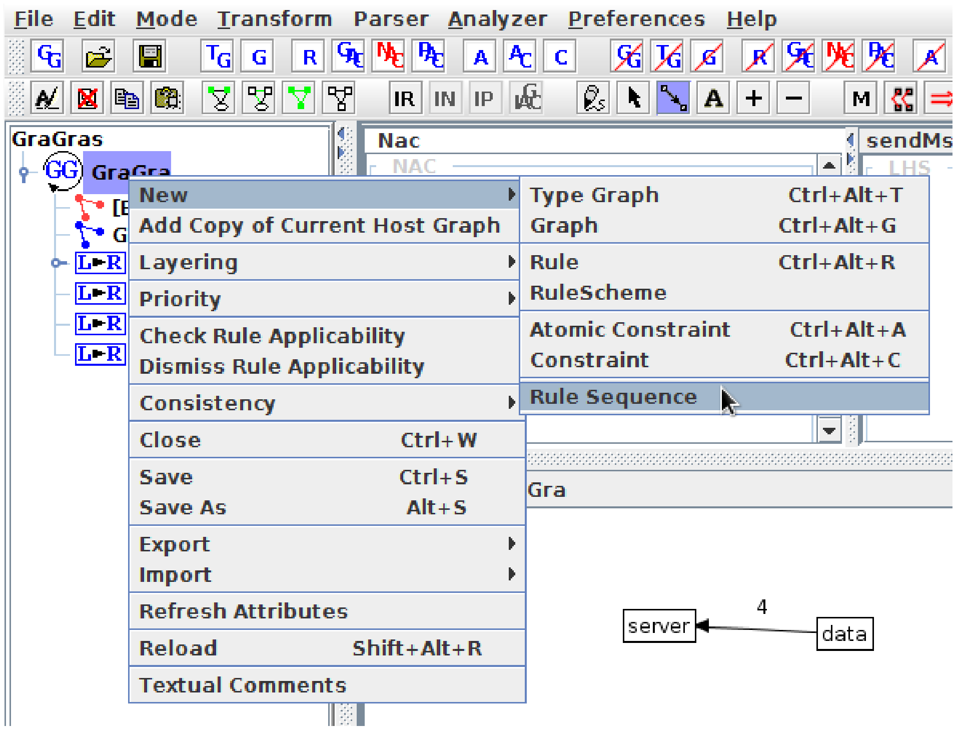
first \ sec...	1	2	3	4
1 sendMsg	0	4	0	0
2 getData	0	0	6	0
3 receiveMsg	0	0	0	2
4 deleteMsg	0	0	0	0

After clicking on the cell, AGG will display the corresponding rules, as well as graphs characterizing the conflicts or dependencies. You may need to rearrange the graphs in order to interpret them. When checking the one conflict between `deleteMsg` and itself, for example, AGG will show the following graphs, indicating that the conflict occurs when both applications try to delete the same edge between `data` and `msg`.

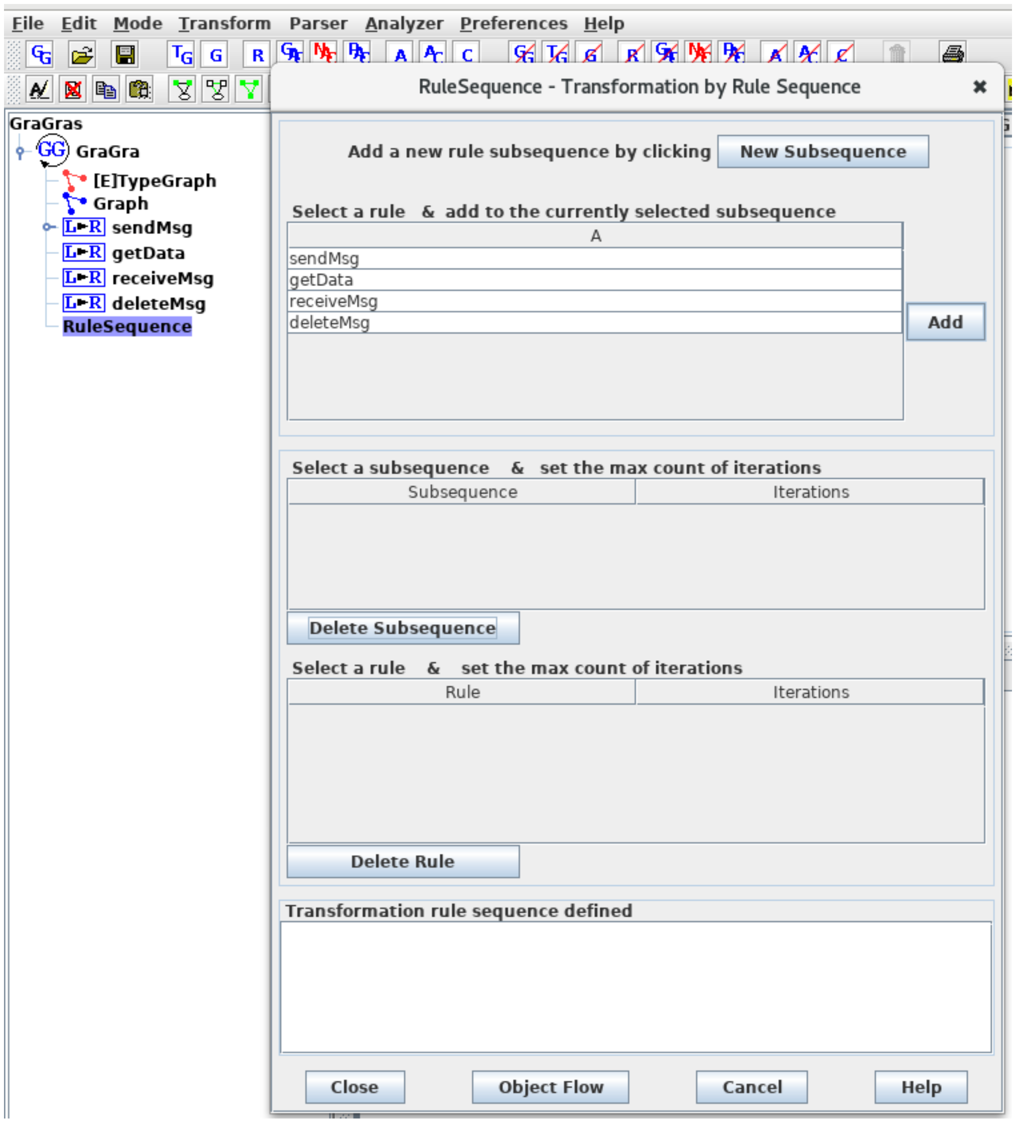


4 Concurrent Rules

Using `AGG`, open the file `grammars/Server/server.ggx`, then right-click on Grammar > New > Rule Sequence.

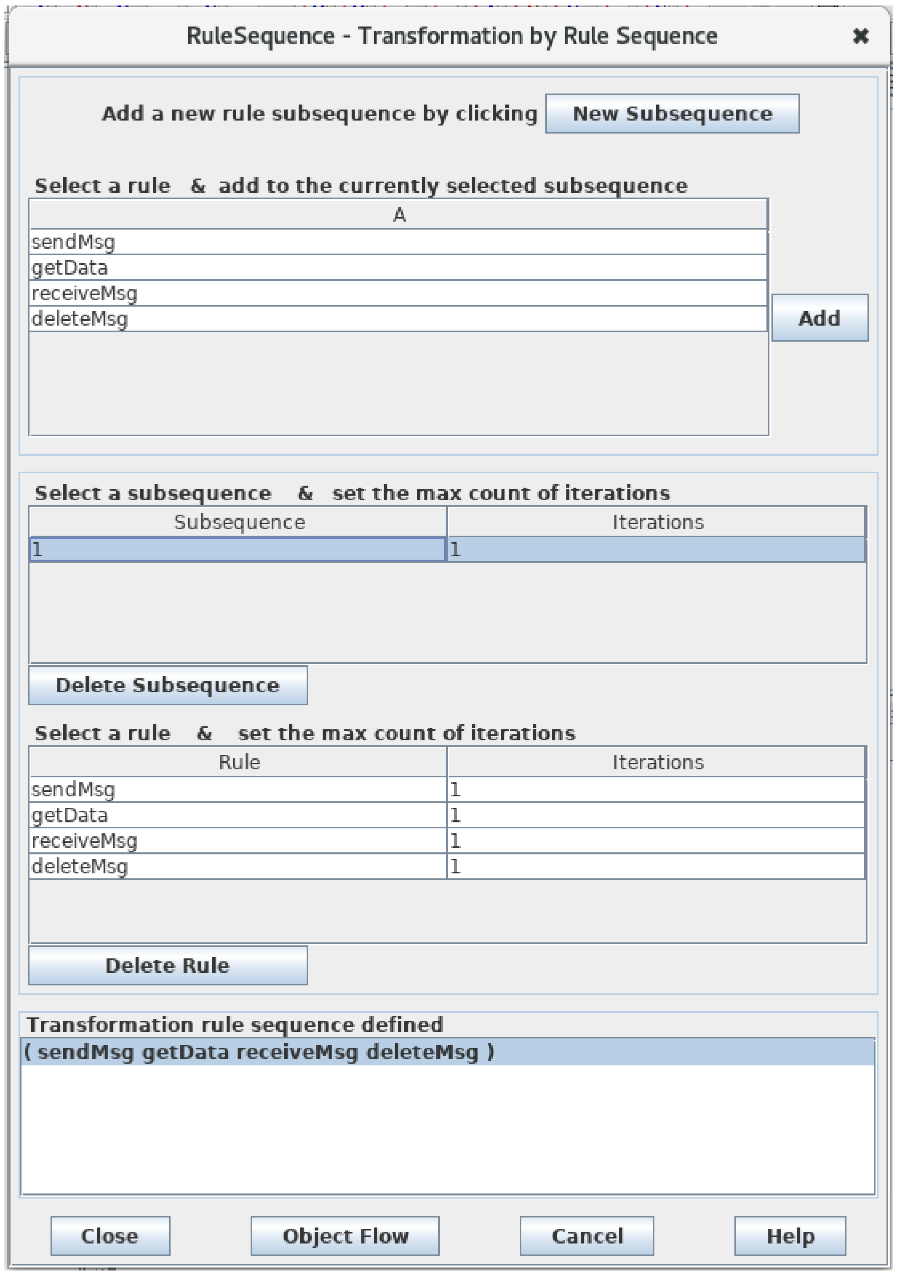


Right-click the rule sequence created below the rules of the grammar, then click on `Show/Edit`, a new pop-up must open.


File Edit Mode Transform Parser Analyzer Preferences Help
GraGras
GraGra
[E]TypeGraph
Graph
sendMsg
getData
receiveMsg
deleteMsg
RuleSequence
RuleSequence - Transformation by Rule Sequence
Add a new rule subsequence by clicking
New Subsequence
Select a rule & add to the currently selected subsequence
A
sendMsg
getData
receiveMsg
deleteMsg
Add
Select a subsequence & set the max count of iterations
Subsequence
Iterations
Delete Subsequence
Select a rule & set the max count of iterations
Rule
Iterations
Delete Rule
Transformation rule sequence defined
Close
Object Flow
Cancel
Help

Click on `New Subsequence` button, a new row will be created at the `Select a subsequence` table. Select the subsequence to add the rules to it.

Once the subsequence was selected, we can add rules to the `Select a rule` table. Select any rule(s) and click the `Add` button. You may add a rule multiples times or change its number of `Iterations` to repeat it. Once the rule sequence is defined, click on close and save the file.


RuleSequence - Transformation by Rule Sequence
Add a new rule subsequence by clicking
New Subsequence
Add
Delete Subsequence
Delete Rule
Close
Object Flow
Cancel
Help

Select a rule & add to the currently selected subsequence

A
sendMsg
getData
receiveMsg
deleteMsg

Select a subsequence & set the max count of iterations

Subsequence	Iterations
1	1

Select a rule & set the max count of iterations

Rule	Iterations
sendMsg	1
getData	1
receiveMsg	1
deleteMsg	1

Transformation rule sequence defined

(sendMsg getData receiveMsg deleteMsg)

Now, we can use `verigraph` to compute all the concurrent rules for this rule sequence, using the following command:

```
$ verigraph concurrent-rule -o output.ggx \
  grammars/Server/server.ggx
```

Note that this will generate *all* concurrent rules, which may be a lot (in this particular case 1202 rules), and may take quite some time. We are currently working on using graph constraints and multiplicity to improve the generation of concurrent rules, reducing the number of spurious rules that are generated.

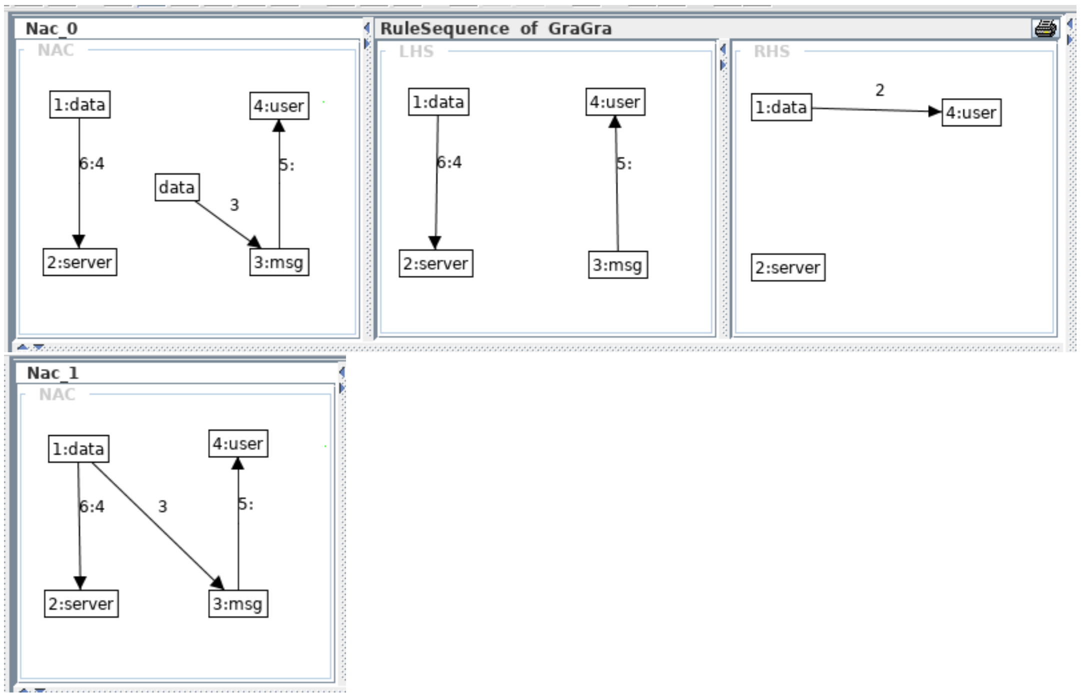
The generated rules are also based on the restriction that matches are always monomorphic. In order to account for arbitrary matches, use the flag `--all-matches` after the `verigraph` command.

4.1 Maximizing Interaction between Rules

The generated concurrent rules allow for different degrees of interaction between rules, including a parallel rule (where the matched subgraphs are completely disjoint). Sometimes, however, only the concurrent rules with maximum interaction are desired, which may be obtained by the following command:

```
  $ verigraph concurrent-rule --max-rule -o output.ggx \
    grammars/Server/server.ggx
```

Then open the `output.ggx` file in `AGG`, which should have a new rule that looks like this:

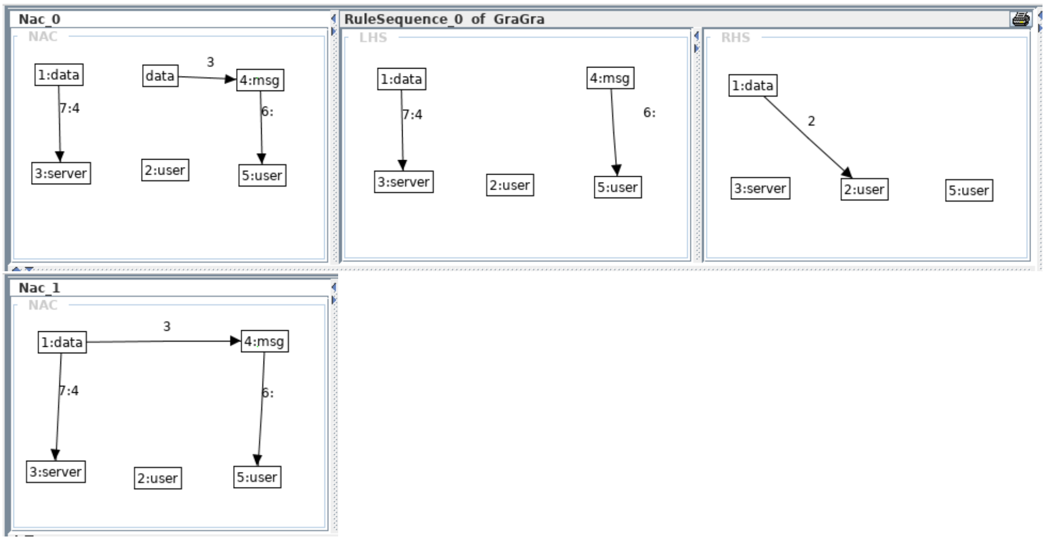


4.2 Dependent rules

Sometimes, maximizing the interaction between rules is an excessively strong restriction. An alternative is generating concurrent rules based on the dependencies between the subsequent rules of a rule sequence. In order to generate such concurrent rules, run:

```
$ verigraph concurrent-rule --all-rules --by-dependency -o \
        output.ggx grammars/Server/server.ggx
```

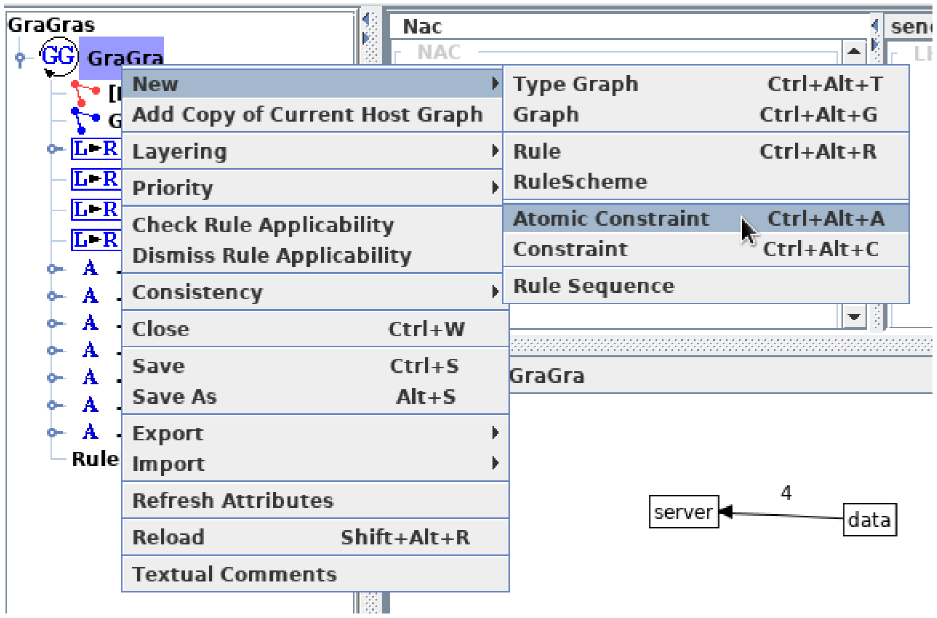
The output file must have two new rules, the same one as the previous example, and another that looks like this:


Nac_0
NAC
1:data
data
3
4:msg
7:4
6:
3:server
2:user
5:user
RuleSequence_0 of GraGra
LHS
1:data
4:msg
7:4
6:
3:server
2:user
5:user
RHS
1:data
2
3:server
2:user
5:user
Nac_1
NAC
1:data
3
4:msg
7:4
6:
3:server
2:user
5:user

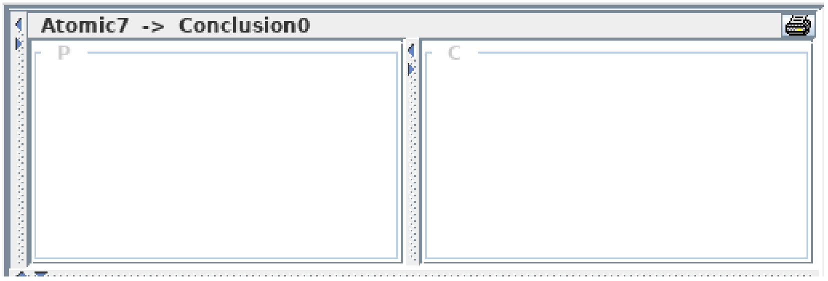
4.3 Constraints

It is possible to use positive and negative atomic constraints to restrict the generation of concurrent rules to more significant ones. This is still a *beta* feature that works only with concurrent rules.

To add new atomic constraints to a Grammar using `AGG`, right-click on Grammar > New > Atomic Constraint.



Click on the newly created atomic constraint to add elements to its *Premise* and *Conclusion* graphs. Notice that *Verigraph* uses the name of an atomic constraint to interpret it as positive or negative: negative constraints have to start with an – (hyphen), otherwise they are treated as positive.



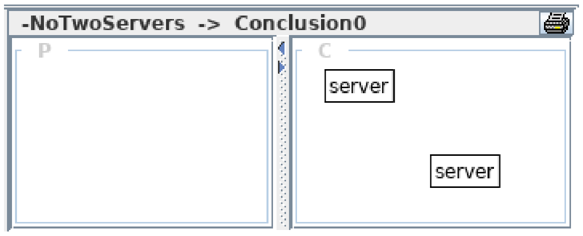


Figure 1: Negative atomic constraint forbidding the existence of more than one server.

Important: Although AGG supports adding multiple conclusion graphs to a constraint, Verigraph doesn't.

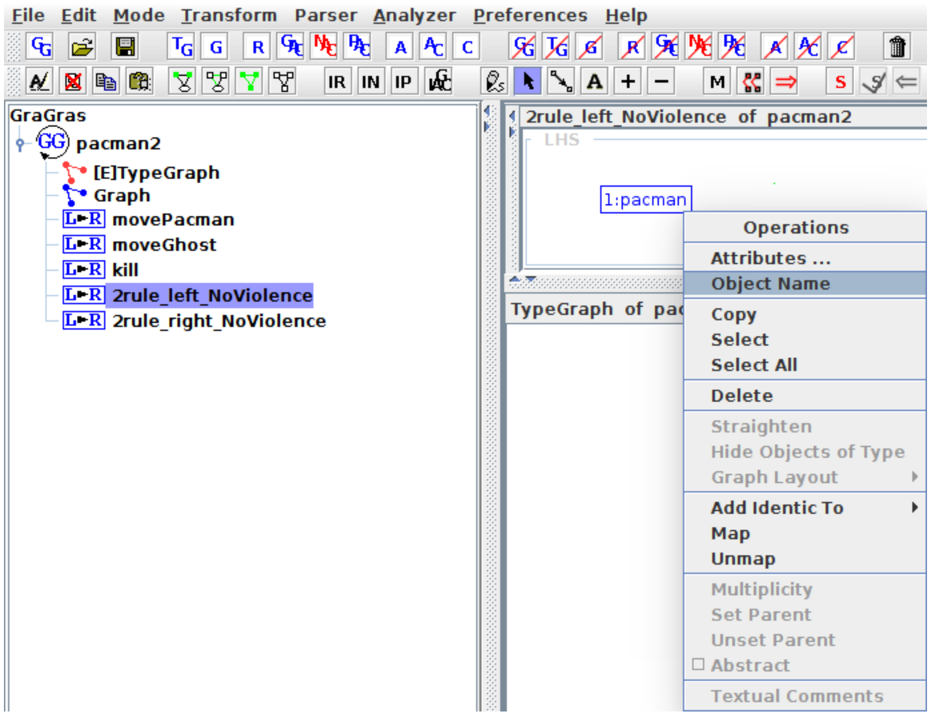
To enable *Verigraph* to use constraints in order to cut off invalid concurrent rules, add the `--use-constraints` flag before the concurrent rule command. For example:

```
$ verigraph --use-constraints concurrent-rule --all-rules \
  --by-dependency -o output.ggx grammars/Server/server.ggx
```

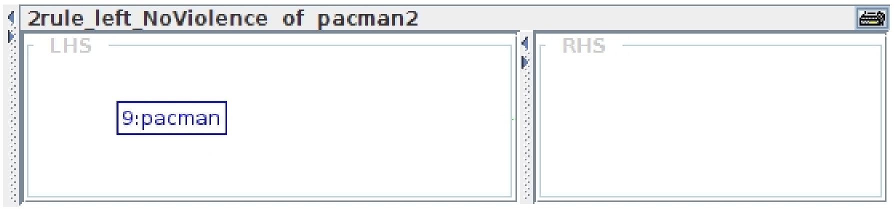
It can be combined with any previous concurrent rule generation strategy. Note that combining constraints and dependency-induced rules will *not* generate a complete set of rules, since any dependencies caused by the constraints are not not yet calculated by verigraph.

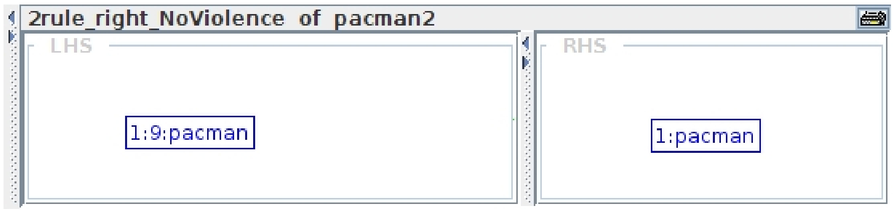
5 Second Order

Although AGG is used for creating the input to `verigraph`, it does not support second order productions. In order to specify such productions in AGG, we simulate them by using two first order rules, following a naming convention and linking their elements with *Object Names*.



The Figure above shows how to set an *Object Name* in a graph element (only numbers are accepted). The Figure below is a second order production simulated in AGG, the two *pacman* nodes linked have the number 9 in their *Object Name*.

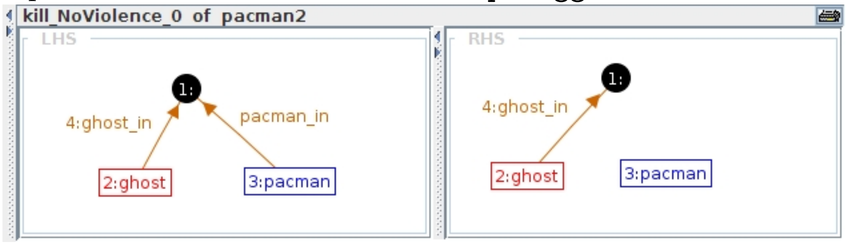




The grammar in `grammars/Pacman/pacman2.ggx` simulates a simple Pacman game. This grammar contains the second order production above. Running the command below will create a new grammar file where all second order productions are applied once on all possible matches of the first order productions.

```
$ verigraph snd-order -o output.ggx grammars/Pacman/pacman2.ggx
```

The new production below must be on `output.ggx` file.



5.1 Inter-level Analysis

Two new kind analysis arise from inter-level interaction between productions. These algorithms are being studied, and their implementations are not considered entirely finished, also it lacks of a support GUI. Despite of it, a log with quantitative information is generated running the command below.

```
$ verigraph analysis --snd-order --all-matches \
        grammars/Pacman/pacman2.ggx
```

6 Contact Information

API internal documentation :
`https://verites.github.io/verigraph-docs/`

Reporting issues with *Verigraph* or its tutorial :
`https://github.com/Verites/verigraph/issues`

Authors' Information :

- Andrei Costa: `https://inf.ufrgs.br/~acosta/`
- Guilherme Grochau Azzi: `https://inf.ufrgs.br/~ggazzi/`
- Jonas Santos Bezerra: `https://inf.ufrgs.br/~jsbezerra/`
- Leila Ribeiro: `https://inf.ufrgs.br/~leila/`
- Leonardo Marques Rodrigues
- Rodrigo Machado: `https://inf.ufrgs.br/~rma/`