The *Hoare-fol* Tool

Maxime Folschette

Univ. Lille, CNRS, Centrale Lille, UMR 9189 – CRIStAL – Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

December 17, 2019

Abstract

This document presents the tool named "Application of Hoare Logic and Dijkstra's Weakest Proposition Calculus to Biological Regulatory Networks Using Path Programs with Branching First-Order Logic Operators" or *Hoare-fol* for short. This tool consists in an implementation of the theoretical work developed in [Bernot et al., 2019] and contains the following features: (1) computation of the weakest precondition of a Hoare triple, (2) simplification of this weakest precondition using De Morgan laws and partial knowledge on the initial state, and (3) translation into Answer Set Programming to allow a solving of all compatible solutions.

1 Introduction

1.1 Biological Regulatory Networks

Algebraic models [Kauffman, 1969, Thomas, 1973, de Jong, 2002] are noteworthy in the field of systems biology for their ease of use. Indeed, contrary to other formalisms such as ordinary differential equation-based models, their complexity remains very low as they do not require to compute an analytical solution. Furthermore, they require much less parameters to function, meaning that they are of great help when too many system parameters are unknown, while still yielding results on the modeled system's behavior.

However, less parameters does not mean no parameters. As a consequence, finding one or several acceptable sets of parameters can still be a challenge, especially if the model is big and that this task cannot be tackled by hand.

The focus of this work is on *Thomas' formalism*, which is typically used to represent Biological Regulatory Networks (BRNs) consisting in interacting components such as genes, proteins, external influences... Formally, it takes the form of a graph in which nodes model components with discrete expression levels and edges model the interactions between these components. More precisely, a specific extension of this formalism is considered, featuring hyperarcs labeled with logic formulas and called *multiplexes*, that are useful to reduce the number

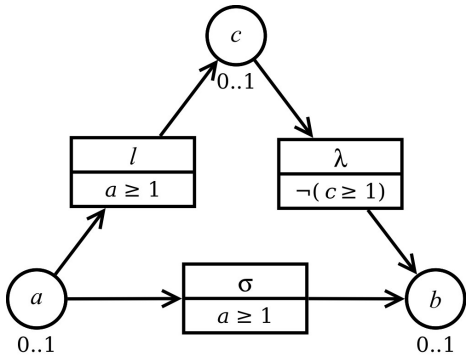

Figure 1: Toy example of [Bernot et al., 2019] representing an incoherent feed-forward loop.

of parameters [Khalis et al., 2009]. An example of such a graph is given in Figure 1.

1.2 Parameters in Thomas' Formalism

Yet, mutiplexes are not sufficient to specify some aspects of the dynamics:

- how interactions play together when several of them point to the same node (that is, which logical gate is used),
- in the case of multi-valued models, how "strong" an interaction is (for instance, does it attract the component to level 1 or to level 2),
- in the case where it is not specified in the graph, whether an interaction "pulls" a component up (activation) or down (inhibition).

To represent this information, parameters for Thomas' formalism were proposed in [Snoussi, 1989] and are now considered as part of the formalism. They are often denoted with the letter k and are uniquely characterized by a couple (v, ω) where v is the variable it refers to and ω is the set of its active predecessors. In other words, when the active predecessors of v (that is, having an effective influence on v) are exactly the set ω, then this variable v is attracted towards the expression level $k_{v,\omega}$.

A set of parameters covering the whole model is called a *parametrization*, and allows to compute, in each possible state, the global "focal point" towards which the model is dynamically attracted. A parametrization is equivalent to a complete set of logic gates (and, or, etc.) between interactions towards the same node, and can also be equivalently translated into an activation function that takes the current state as input, and outputs the set of possible next states. The parametrization representation is preferred here because it is central to Thomas' formalism and has been tailored to this kind of representaiton.

1.3 Context of the *Hoare-fol* tool

The present work relies on the work developed in [Bernot et al., 2019] which aims at providing a way to filter out unwanted parametrizations for a given model, based on known possible dynamical behaviors. This work relies on the classical Hoare logic [Hoare, 1969] by adapting it to Thomas' formalism: imperative programs become dynamical path programs (that represent a possible dynamical behavior of the model), and pre- and postconditions on the program's variables become conditions on the initial and final states and on the parametrization. It also relies on Dijkstra's weakest preconditon calculus [Dijkstra, 1978] to compute such information.

The rest of this docuemnt describes an implementation of this work under the form of the tool *Hoare-fol*. This implementation is written in OCaml and allows an export of the results to Clingo's Answer Set Programming (ASP) [Gebser et al., 2016, Baral and Gelfond, 1994] in order to enumerate all solutions.

This tool is a follow-up to the work started in [Folschette, 2011], where two unsuccessful approaches were taken:

- using Coq to formally prove the new Hoare logic developed,
- using OCaml with functions to encode formulas (pre- and postconditions).

Both methods were not suited for weakest precondition calculus and manipulation, thus giving impractical or partial results. This document, however, proposes a working proof-of-concept of such an implementation.

2 Implementation

The general idea of this implementation is to represent all conditions (pre- and postdconditions, and conditions of multiplexes) and dynamical path programs as symbolic trees in OCaml, by defining constructs for each type of node. This representation allows to easily manipulate them in order to perform precondition calculus, simplification, translation to ASP, and so on.

Note however that in the current version of the implementation, all information (model, conditions, processings) must be provided as OCaml definitions in the main program[1].

2.1 Model, Program and Formula Definitions

First, the Thomas model takes the form of two lists representing the component nodes (`vars`) and multiplex nodes (`mults`) with all related information (predecessors, conditions, etc.). Each element of these lists are couples where the first element is a string giving the name of the component (variable or multiplex)

[1] Which is of course neither developer-friendly nor a good practise, but this tool only intends to be a proof-of-concept.

$\varphi :==$	`MPropConst`(b)	Constant proposition: b is either True or False
	| `MPropUn`(n, φ)	Unary proposition: n is the negation
	| `MPropBin`(o, φ, φ)	Binary proposition: o is a connective ($\wedge$, $\vee$, ...)
	| `MRel`(c, ψ, ψ)	Comparison: c is a comparator ($=$, $>$, $\geq$, ...)
	| `MAtom`(v, i)	Atom on a variable: means ($v \geq i$)
	| `MMult`(m)	Atom on a multiplex: recalls the formula of m
$\psi :==$	`MExprBin`(o, ψ, ψ)	Arithmetic operation: o is an operator ($+$ or $-$)
	| `MExprConst`(i)	Constant: i is an integer

Figure 2: OCaml grammar for multiplex formulas (φ) and multiplex arithmetic expressions (ψ).

```
 let vars = [("a", (1, [])) ;
             ("b", (1, ["lambda" ; "sigma"])) ;
             ("c", (1, ["l"]))] ;;
let mults = [("l", MAtom("a", 1)) ;
             ("lambda", MPropUn(Neg, MAtom("c", 1))) ;
             ("sigma", MAtom("a", 1))] ;;
```

Figure 3: OCaml representation of the toy example of [Bernot et al., 2019].

and the second is the information attached. In the case of `vars`, the second element is also a couple where the first element is the upper bound of the variable (integer) and the second is the list of precedessor multiplex names (list of strings). Regarding `mults`, the second element is the *multiplex formula*, which follows the grammar given in Figure 2 and which itself contains information about its predecessors (variables or multiplexes). For instance, the model of Figure 1 taken from [Bernot et al., 2019] is represented in OCaml as the listing of Figure 3.

Then, general-purpose *formulas* can be defined with another grammar defined in Figure 4 which is close to the multiplex grammar. Such formulas will be used to define postconditions in order to perform weakest precondition calculus, but they could also be used to describe other kinds of conditions. As a matter of fact, they are also used to express conditions and invariants in `If` and `While` control flow instructions. For instance, the postcondition ($a = 1 \wedge b = 0$) can be expressed with:

```
let my_post = PropBin(And,
                Rel(Eq, ExprVar "a", ExprConst 1),
                Rel(Eq, ExprVar "b", ExprConst 0)) ;;
```

Finally, an imperative *path program* can be defined with the grammar given in Figure 5 which copies classical imperative program instructions (assignments) and control flow (`If`, `While`) but also adds descriptions for possible ($\exists$) and mandatory ($\forall$) dynamical branchings. As an example, the path program ($b+; c+; b-$)

Φ :==	`PropConst`(b)	Constant proposition: b is either True or False
	| `PropUn`(n, Φ)	Unary proposition: n is the negation
	| `PropBin`(c, Φ, Φ)	Binary proposition: o is a connective ($\wedge$, $\vee$, ...)
	| `Rel`(r, Ψ, Ψ)	Comparison: c is a comparator ($=$, $>$, $\geq$, ...)
	| `FreshState`(Φ)	Formula on a fresh set of variables
Ψ :==	`ExprBin`(o, Ψ, Ψ)	Arithmetic operation: o is an operator ($+$ or $-$)
	| `ExprVar`(v)	Valuation of variable: the value of variable v
	| `ExprParam`(v, ω)	Valuation of parameter: the value of parameter $k_{v,\omega}$
	| `ExprConst`(i)	Constant: i is an integer

Figure 4: OCaml grammar for formulas (Φ) and arithmetic expressions (Ψ) to be used in general-purpose conditions (preconditions, postconditions and control flow instructions).

Π :==	`Skip`	Does nothing
	| `Set`(v, i)	Assignment: $v \leftarrow i$
	| `Incr`(v)	Increment: $v+$, i.e., $v \leftarrow v + 1$
	| `Decr`(v)	Decrement: $v-$, i.e., $v \leftarrow v - 1$
	| `Seq`(Π, Π)	Instructions sequence
	| `If`(Φ, Π, Π)	If-then-else conditional
	| `While`(Φ, Φ, Π)	While loop: requires a condition and a loop invariant
	| `Forall`(Π, Π)	Dynamical branching: both behaviors are possible
	| `Exists`(Π, Π)	Dynamical branching: at least one behavior is possible
	| `Assert`(Φ)	Assertion: the formula is true at this point

Figure 5: OCaml grammar for imperative path programs (Π). The symbol Φ refers to the formulas grammar defined in Figure 4.

can be expressed with:

```
let my_prog = Seq(Seq(Incr "b", Incr "c"), Decr "b") ;;
```

Note that in the main OCaml file, the model specification should be written just after the multiplex formula grammar definition, while the formulas and path programs to process should be defined at the end of the file.

2.2 Useful Functions

Taking into consideration the program and the post-condition, the weakest precondition can be computed with the `wp` function:

```
let my_wp = wp my_prog my_post ;;
```

A simplification can be applied on any formula with the `simplify` function. If an initial state and a parametrization are (partially) known, one can also "refine" (that is, strengthen) the weakest precondition with the same function:

```
let simpl_wp = simplify my_wp known_vars known_params ;;
```

where `my_wp` is the weakest precondition (or any other formula to simplify), and `known_vars` and `known_params` are association lists giving the known values of any number of variables and parameters. If no such infomation is given, empty lists (`[]`) are to be provided. In any case, the `simplify` function replaces all variables and parameters given in these lists by their provided values, and attempts to perform basic simplifications on the formula, following De Morgan's laws.

At any point, functions are provided to translate a formula (`string_of_formula`), an arithmetic expression (`string_of_expr`) or an imperative path program (`string_of_prog` and `string_of_prog_indent`) into a pretty-printable string.

Finally, one can translate a formula (typically, the simplified and refined weakest precondition) into Answer Set Programming (ASP) that can be read by Clingo by using function `write_example`:

```
write_example my_wp "file.lp" ;;
```

This translation is made by creating an ASP atom for each node of the OCaml representation of the formula. This atom is used in rules such that it reflect its semantics. For instance, consider the conjunction $f = a \wedge b$ between some subformulas a and b, which ASP representations are atoms `atom0` and `atom1`. This fomrula would be translated into another atom, say `atom2`, and the conjunction would be encoded by the ASP rule:

```
atom2 :- atom0, atom1.
```

Arithmetic expressions are also translated into their ASP equivalent, while variables and parameters are each assigned to an ASP variable.

Note that the ASP variables that represend parameters are labeled with integers rather than with the explicit names of the resource set ω. In order to obtain the correspondence between the ASP variable names and the original parameters, one can use the `asp_params` function.

See the "Sandbox" part at the end of the main OCaml file for examples on how to use theses functions to obtain results on the model example.

3 Contents and Usage

The *Hoare-fol* tool is freely available at `https://gitlab.cristal.univ-lille.fr/mfolsche/hoare-fol` under the MIT license[2].

The tool can be run in a Unix compatible terminal. Pease refer to the `README.txt` file for information on the requirements and how to run the main file.

The `main.ml` OCaml file contains the main program with somme example applications on the model of Figure 3. It requires OCaml 4.03 to be executed, but the latest version[3] is recommended. The command line to run this file is:

```
ocaml main.ml
```

[2] `https://opensource.org/licenses/MIT`

[3] OCaml version 4.09 at the time of writing

If ASP files are produced by the execution, they can be fed to Clingo[4] with the following command:

```
clingo 0 file.lp
```

Note that the command line option `0` means "enumerate all solutions". It can be replaced by a non-null integer to indicate the maximum number of solutions to enumerate

The provided script `run-all.sh` allows to run all `.lp` files with Clingo and store the results in `.lp.out` files:

```
bash run-all.sh
```

4 Limitations

This implementation comes as a proof of concept, and as such still has a number of limitations.

The biggest theoretical limitations are linked to the `While` loops that are rather difficult to express, and which support is limited in the current version of this tool:

- An explicit loop invariant has to be provided for the `While` loops. However, [Bernot et al., 2019] propose a method to automatically infer a weakest invariant with the following approach:
  - Start with the most general invariant.
  - Run the loop and remove values that lead out of boundaries.
  - Repeat until reaching a fixpoint.

  Since values are finite (variables take bounded discrete values), this is ensured to end.
- The weakest preconditions of `While` loops are expressed as formulas in a special context (`FreshState`, defining a "fresh" set of variables) which is currently not explored nor simplified buy the `simplify` function. The simplifications should also apply to these formulas, probably with the same simplification rules, but by taking care of not performing refining on variables in such a context.

There also are technical limitations regarding the ASP output:

- The output of Clingo can be difficult to read, as variables are all encoded with dummy names. The `asp_params` outputs the correspondence between ASP and model variables, bot does not provide pretty-printing nor replace one with the other in the output. A more explicit encoding could be found to ease direct reading of the solutions.

[4] Produced scripts are intended for Clingo 5. This feature has not been tested with Clingo 4 although the syntax should be compatible. It is compatible with Clingo 3, but it is advised to comment out the `#hide.` directive in the produced scripts to hide uninteresting atoms.

- The Clingo solving can be really long for some formulas, especially if there are a lot of solutions. This limitation seems hard to fix; working on the formula instead of on the set of solutions seems to be the best alternative in this case.

Finally, there also are obvious limitations on the source code itself:

- Both model and processings have to be hard-coded in the main file, and at specific locations. A parser should be added to load a model from a file, or the main file without examples should be turned into a module.
- Functions related to Hoare logic should be purified (they are currently closures on `vars` and `mults`, which partly causes the previous limitation).

5 Conclusion

This paper presents an implementation of [Bernot et al., 2019] which applies Hoare logic to Thomas' formalism in order to infer constraints on the model's parameter values. It relies on a symbolic representation of logic formulas and imperative programs in order to compute the weakest precondition of a given couple of program and postcondition. It is written in OCaml and allows an ouput of the formulas in ASP (compatible with Clingo 5) to enumerate solutions. Although there are theoretical and technical limitations, especially when `While` loops are involved, or regarding hard-coded features, this work only aims at being the basis of other works that could require such a framework. This has already been the case with [Behaegel et al., 2017] which re-uses its main concepts, and applies them to a hybrid extension of Thomas' formalism.

References

[Baral and Gelfond, 1994] Baral, C. and Gelfond, M. (1994). Logic programming and knowledge representation. *The Journal of Logic Programming*, 19-20:73 – 148. Special Issue: Ten Years of Logic Programming.

[Behaegel et al., 2017] Behaegel, J., Comet, J.-P., and Folschette, M. (2017). Constraint Identification Using Modified Hoare Logic on Hybrid Models of Gene Networks. In Schewe, S., Schneider, T., and Wijsen, J., editors, *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, volume 90 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:21, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Bernot et al., 2019] Bernot, G., Comet, J.-P., Khalis, Z., Richard, A., and Roux, O. (2019). A genetically modified hoare logic. *Theoretical Computer Science*, 765:145 – 157. Formal Verification and Static Analysis of Molecular Devices and Biological Systems.

[de Jong, 2002] de Jong, H. (2002). Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, 9(1):67–103. PMID: 11911796.

[Dijkstra, 1978] Dijkstra, E. W. (1978). *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs*, pages 166–175. Springer New York, New York, NY.

[Folschette, 2011] Folschette, M. (2011). Application de la logique de Hoare aux réseaux de régulation génétique avec multiplexes. Master thesis, École Centrale de Nantes, Nantes.

[Gebser et al., 2016] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Wanko, P. (2016). Theory Solving Made Easy with Clingo 5. In Carro, M., King, A., Saeedloei, N., and Vos, M. D., editors, *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, volume 52 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:15, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Kauffman, 1969] Kauffman, S. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437 – 467.

[Khalis et al., 2009] Khalis, Z., Bernot, G., and Comet, J.-P. (2009). Gene Regulatory Networks: Introduction of multiplexes into R. Thomas' modelling. In Amar, P., Képès, F., Norris, V., and Bernot, G., editors, *Proc. of the Nice Spring school on Modelling and simulation of biological processes in the context of genomics*, pages 139–151. EDP Sciences.

[Snoussi, 1989] Snoussi, E. H. (1989). Qualitative dynamics of piecewise-linear differential equations: a discrete mapping approach. *Dynamics and stability of Systems*, 4(3-4):565–583.

[Thomas, 1973] Thomas, R. (1973). Boolean formalization of genetic control circuits. *Journal of Theoretical Biology*, 42(3):563 – 585.