Comhordú in Coq: Documentation and Project Plan

May 8, 2015

The aim of this document is to give an overview of the remaining work left to do for the project of proving Comhordú correct in the Coq theorem prover. The document provides a high-level view of the state of the project as it is, a plan towards completion, some general coding guidelines and possibilities for future extensions.

1 Summary of Existing Model

To begin with, Table 1 summarises the high level statistics summarising the content of the model.

Total lines of code	13459	Proved theorems	693
Completed Definitions	259	Inductive definitions	92
Tactics	225	Axioms	28
Parameters	5	Admitted theorems	226
Conjecture	28		

Table 1: High Level Statistics of Coq Model

Of this content, the parameters and admitted theorems constitute the remaining work to do. Parameters are incomplete definitions, much like stub functions that can be written when developing software. Admitted theorems are those whose proofs have yet not been completed, or whose proofs have been scrapped due to some modifications rendering them obsolete. Conjectures are theorems statements that are assumed without proof because they are deemed obvious. It is not a priority to prove conjectures, though it would be desirable to prove them for the sake of completeness.

The rest of the content is complete. Proved theorems are fully defined, albeit in terms of lower level results which themselves may not be fully defined. Completed definitions are similar to proved theorems. In fact, by the Curry-Howard isomorphism, they almost the same- except that Coq makes a distinction between sets and propositions. Inductive definitions are also complete, and differ from other definitions only in their definition style, which uses constructors.

Tactics are scripts to help reduce the verbosity of proofs. Axioms are assumptions that are not supposed to be defined i.e. black box functions that are purposely left uninterpreted for the sake of generality.

2 Admitted Labels

The 226 remaining theorems to be proved are each classified by a label. The labels are a notational convention that has been adopted for this project, rather than a Coq primitive. They are included after the "Admitted" keyword as a comment e.g. "Admitted. (*C*)" will appear at the end of an admitted theorem tagged with the "C" label. The labels and their meanings are described in the following.

C This is taken to mean that an assumed result actually has to be changed before it can be proved. Usually the change is minor e.g. adding a simple hypothesis. Once the change is made, the proof should be possible. After the modified proof is completed, any theorems depending on this proof will need to be adjusted. That is, a proof referencing the theorem will need to use it in its new form. Usually the modifications to higher-level results are minor, and usually a single theorem does not have a lot of results depending on it directly. Sometimes a higher-level proof itself must be changed to accommodate a change at a lower level, and the change may propagate up a few levels. From experience, such changes tend to be minor.

D This means that an assumed result depends on the definition/proof of another function/result before it is proved itself. The result/function upon which this result depends is written after the "D".

R This means that the proof needs to be redone in light of a switch to bisimulations for state predicate encodings. An older version of the model encoded state predicates in a different way, and all proofs tagged with "R" rely on this old version of state predicates, and associated tactics. The old proofs are commented out and the "Qed" is replaced with "Admitted". It is likely that the basic structure of such proofs can remain the same, with some adjustments made for the new version of state predicates. It is also likely that these adjustments will share similarities across different proofs. For example, the repeated use of an out-of-date tactic, state_pred_elim, across many proofs, can be replaced by the repeated use of a modified version of this tactic. These results are not a priority as they have already been proved with the different definition of state predicates. It is unlikely that the switch to different versions of state precicates will change their truth value.

#... A number of theorems are hashtagged with some label. There are many different labels, each denoting a category of similar theorems. Theorems

tagged with the same label can be proved in much the same way. It seems worthwhile to develop automatic tactics to aid with the proofs of these theorems: since the there are many similar theorems, tactics can be reused. This saves time and effort repeating similar or identical sequences of steps in distinct but related proofs. The list of hashtags will be discussed later in the document in the context of tactics.

1-10 The remaining theorems are numbered between 1 and 10 to indicate a difficulty level for the proof. The theorems numbered 1 are the most difficult and those with 10 are the easiest. Of course, these are just heuristic predictions; grading a theorem's difficulty before it is proved is not a hard science. It is possible that some of these theorems could be grouped together and categorised under a common tag. Regular surveys of the model can be useful in developing such categorisations, among other things, and are discussed later.

Table 2 gives the number of remaining theorems in each of the categories mentioned above i.e. C, D, R, # and numbered.

C	D	R	#	Numbered
3	4	13	45	162

Table 2: High Level Statistics of Coq Model

3 Project Plan

The goal of this project is to prove all the remaining theorems and define all parameters. If time allows, the conjectures can be proved, though they're not a priority as they've been deemed obvious. The various categories of theorems are listed in the previous section

Most of the theorems are simply marked with a number from 1-10. Roughly speaking, theorems marked from 1-3 require a good deal of higher-level thought and planning, can involve about 100-200 lines of tactics, and require the creation of 5-10 auxiliary results. Theorems between 4-6 can often be proved without much thought, but can still be long and often require about 5 auxiliary results. The theorems labelled 7-10 should be straightforward enough.

The theorems marked with C and R require changes to be made before they are completed. The changes to a C theorem will probably be small e.g. adding a hypothesis. The degree of change needed in an R theorem depends on how heavily the theorem relies on the definition of state predicates.

The theorems marked D require something else to be proved or defined first. In order to prove one of these, it is necessary to first prove or define the specified result.

The remaining theorems are all categorised for tactic application. The various categories are discussed in Section 3.1.

3.1 Tactic Development

For the tagged theorems, tactic development is the proposed route. The idea is to develop a general tactic or suite of tactics to completely solve all the results in a category, or at least greatly reduce the effort of proving them. There is a trade-off here between the cost of developing tactics and the benefit of reusing them. Sometimes it is not worth developing a tactic, even if there are multiple theorems in a particular category: often one proof can simply be copied and pasted across to another theorem, after some renaming. The following lists the existing categories (hashtags) and discusses possible tactics suitable to each category.

Delay preservation It can be shown that state predicates are preserved over a delay: both forwards and backwards. This leads to a suite of tactics to be developed, at all levels. Existing tags are: #delay-pres, #del-pres-net and #del-pres-bkwd-net. More tags are probably appropriate for different levels and/or directions of preservation.

#state-pred-elim This tactic derives a contradiction from the assumption that two incompatible state predicates hold. The predicates can hold over a software term, an entity or a network. there are four levels of tactics to develop: proc, prot, ent, net. The first is for a raw software term. The second is for a term lifted to a protocol triple. The third is for an entity. The fourth is for a network. It should be possible to build higher-level tactics from lower-level ones.

#inter-component These results all involve relationships between the various sub-components in an entity. Usually it is a relationship between the software component and either the interface or mode state component. These results are diverse, and it doesn't seem likely that there is one "master tactic" to prove them all in . What does seem possible though, is that a suite of tactics could be written that are tailored to basic inter-component relationships. The tactics should be simple enough to be resuable, but complex enough to be worthwhile developing, as opposed to simply attacking the theorems one by one manually.

#timeSplit-timedList A number of theorems on the various flavours of timed lists are the similar, stating that a delay can be split into two. It seems like an opportunity for the application of general purpose tactics.

#complexNetLift Various complex state predicates. It may be that slightly different tactics are needed for the different state predicates. For this reason, there are the more specific tags #nextSinceStateNetLift, #tfsNetLift and #notifBadNetLift.

#fwd-track Forward tracking: match successor state given some source state. Also need a tactic for back tracking.

Lifting Various tactics could be written for lifting results from one level to another. Usually, results are lifted from either the process level to the entity level, or from the entity level to the network level. Perhaps some general lifting tactics are possible, as well as the more specific tatctics listed as follows: #back-track-lift-ent, #fwd-track-lift-ent, #back-track-lift-ent, #inter-component-lift-net, #lift-track-net.

#strong Back and forward tracking where not only the states are matched but the actions and parameters also.

3.2 Non-Priority Tasks

Some tasks remain that could be done but are not a priority. Naturally, it makes sense to leave these tasks until the main work is done. Still, some tasks actually may be worthwhile e.g. they may result in neater code which is more workable. Also, it may be the case that the coder becomes stuck figuring out something in the main development. In the meanwhile, it is useful to have tasks like these to fall back on while a more important problem is being figured out. The tasks are listed in the following.

Garbage Collection There are a number of definitions, theorems, comments etc. that do not serve a higher purpose. There are a number of possible reasons for the existence of such objects e.g. a high-level theorem had to be changed and it now no longer uses a lower level result. Leaving these unused objects in the development is harmless, though they do clutter up the space and make the code slightly less readable. A solution would be to either remove these objects altogether or move them out of the way e.g. at the end of the file. In particular, it would be useful to discard any unproved theorems that are not needed at a higher level- this would not only remove clutter but save time in proving them later. However it may not always be clear whether an Admitted result it needed or not. Another possibility would be to remove duplicate results: the same thing proved twice or more, possibly under different names.

Demotion/Promotion Some results that have "theorem" status should really only have "lemma" status and vice versa. Technically, it makes no difference which word is used, but from a human perspective, it is conceptually more elegant to give a result a name appropriate to its difficulty/importance. This is completely a subjective choice, though it would be possible to come up with some metrics guidelines e.g. number of lines of code in a proof.

Renaming things Standardisation of names is of moderate importance. Although the names of theorems can be arbitrarily chosen, it makes sense to choose names that conform to some sort of convention so that they are easier to find later. For example, ending all lifting theorems with "_lift" allows a user to search across all files This saves a coder later

Standardisation Results belonging to a certain category should be standardised in their statement. E.g. if all results take a network and an integer as parameters then the order in which these parameters are stated should be the same across all results. Standardisation makes it easier to write automatic tactis that work across all results. On the flip side, tactics should be written robustly to deal with different variations on a the statement of results.

Re-document This would involve addding or modifying documentation on some results/definitions. It is unlikely that any important objects are undocumented but there may be a small number. Also, it may be necessary to revise some old documentation in light of new developments and knowledge.

Explicit typing Explicit typing improves readability, but is not necessary whenever Coq can infer the types. If this task is to be undertaken, perhaps it would be useful to briefly research if there's an automatic tool for doing this? Another possibility for explicity typing would be to do a search for the object in question and then copy and paste what comes up in the results pane, because that always has types.

4 General Coding Practice

Experience working on this project has led to the following guidelines for good coding practice. These guidelines do not constitute an exact science, but it is my belief that they are useful for saving time and producing neat code. They could alternatively have been referred to as "good coding habits". These are the main ones that come to mind:

Find/replace I use Notepad++ for its find/replace functionality, but an equally powerful tool would do just as well. The key features are a capability for find/replace across multiple files and regular expression matching. This is useful for renaming but mostly for finding things. Coq itself has an in-built search mechanism but it restricted to either standard pattern matching (no regex) against the text within one file only, or searching for a function/proof by name or by type. What can't be done is to search for notations, for occurrences of a function/theorem, or for part of a name. The last point is particularly important. For example,we might be searching for a linking theorem but not know its exact name. Searching for "_link" in notepad++ gives every occurrence of this string within the files. Any theorems with this text in them will be found. Of course, all occurrences of these theorems will also appear, along with anywhere else the text appears, but generally it is easy enough to find the desired object within the results.

General survey Periodically it is good practice to step back from the coding

and survey the model, or parts of it. A survey can involve any of the following:

- Looking for categorisations to group theorems. Grouping a number of similar theorems together opens the door to the possibility of developing and applying category-specific tactics to that group of theorems. Hashtags are used to mark a result as belonging in a certain category. These can be easily found in Notepad++, since the # symbol does not occur much elsewhere in the Coq model (it is used in some rare notations).
- Counting the number of remaining theorems. This allows plans to be made and deadlines to be estimated. It is also useful to count specific categories e.g. how many level 1 theorems are left? How many level 2 theorems are there? And so on.
- Revising the project plan. This may be necessary in light of new theorems that were spawned, or some problem that was discovered. Or perhaps an idea was stumbled upon that would make progress more effiicient. For whatever reasons, revising the plan periodically is useful.

Tidying Periodically, it is useful to tidy the files: neat files are easier to manage, search, read etc. Tidying involves moving and reorganising code, either within a file or across files. It is good practice to re-compile after a tidy- usually moving results results in some minor problem e.g. an extra include is now needed in some files because a result was moved to a different file.

Tactics Development This involves stepping back from "proof mode" and pondering a clever solution to a general problem. A category is chosen- it makes sense to choose the biggest/most important category, though there may be a reason for choosing another category e.g. an idea for a tactic came to mind. The results in the chosen category can be analysed and hopefully a general pattern emerges for their proofs. Then with a bit of brainstorming this general pattern can be turned into one or more tactics. Usually there will be a master tactic and some helpers. The tactics should be written robustly so as to ignore things like the order of hypotheses or their names. A good way to write tactics is interactively: start by proving one theorem in the category and then incrementally break this proof into tactics, testing them as they are developed. The final step with this approach is usually to collect together all the tactics into one master tactic, test it, and then apply it to all the theorems in the category. It is also possible to write tactics for solving more general problems that can be applied across a number of theorems- these can either be specific to the model or completely general (and usable by other Coq users).

Researching Coq The Coq website and mailing list are full of useful information. The website has all the standard tactics and the mailing list has

solutions to more advanced problems. You can also send an email to the mailing list with a specific problem- though it is good manners to search around yourself first before burdening others with your problem.

Use tactics Wherever possible, it is good to use tactics if they reduce the amount of steps in a proof. These can either be tactics from the Coq website or user-defined tactics specific to the model. With practice, tactic use tends to get more sophisticated and proofs get shorter. The only problem with tactics is that they are generally slower the more complex they get. This has not been a real concern so far though.

5 Potential Future Extensions

There are a number of ways in which this model could be extended. This differs from completion of the project, in that completion refers to finishing the existing proofs that remain, towards the high-level proof of protocol correctness, whereas extensions add completely new things to the model. Here are just a few suggestions for future extensions:

- Proof that an unsafe reachable state is possible. This would be useful to demonstrate that the proof of protocol correcntess is not simply trivially true. We want to show that without the protocol, unsafe states are reachable. This would demonstrate a need for the protoocl.
- Axiom Soundness Check. Perhaps it would be worthwhile to manually check the axioms for mutual consistency. This could either be an informal check for obvious absurdities, a rough argument or sketch proof showing consistency, or a formal proof. Perhaps a proof of consistency is even possible in Coq itself, though it's debatable whether or not Godel would allow it.