HAL
open science

A tool for checking CSP||B specifications

Huu Nghia Nguyen, Jean-Pierre Jacquot

To cite this version:

Huu Nghia Nguyen, Jean-Pierre Jacquot. A tool for checking CSP||B specifications. Workshop on Tool Building in Formal Methods - Held in conjunction with the 2nd International ABZ Conference, Feb 2010, Orford (Québec), Canada. inria-00463422

HAL Id: inria-00463422

https://inria.hal.science/inria-00463422

Submitted on 12 Mar 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A tool for checking CSP||B specifications ⋆

Huu Nghia Nguyen, Jean-Pierre Jacquot

LORIA – Nancy Université
Campus Scientifique, BP 239,
F-54506, Vandœuvre lès Nancy, France
{firstname.lastname}@loria.fr

1 Introduction

This paper reports about our experience with building a simple tool to assist us in the verification of CSP||B specifications.

Our primary research goal is methodological: we aim at understanding and modeling how people develop formal specifications. Thus, a language and its supporting environment is simply a tool that specifiers use for a specific purpose. Tools are always designed as a trade-off between range of applicability and efficiency. The most efficient tools often have a narrow purpose. Like carpenters or plumbers, specifiers must carry heavy toolboxes rather than a single do-it-all tool.

An important difference between the tools used for hardware and for software is that, in the latter case, they have very strong and subtle interactions. Actually, they must often be used together at the same time. One specification may require the use of B for managing some complex data structure, of CSP for maintaining some liveness properties, and of UML for validation by the stakeholders. "Gluing tools" are needed.

In a recent specification, in a context of multi agents system, we used CSP||B to model the bevahiors of agents and their communications. While most of the work could be done within language specific tools (FDR2[1] for the CSP part and AtelierB[2] for B part), the verification of the coherence between B and CSP prompted us to design and implement a small assistant. We had at least three reasons:

- Practical use of formal methods entails a bit of intelligent work and a huge amount of boring stuff which requires an attention to all details. Computers, with the proper tools, are good at the second type of work while we human are bad at it; of course, the opposite is true for the first type of work.
- The issue of the coherence proofs between CSP and B is quite subtle and needs a precise understanding. Working on an implementation is a good way to make the fundamental questions pop out.
- We believe that even incomplete tools can be useful: they can leave out some of the intelligent part so long as they take care of all the tedious part. Prototyping is the best way to discover whether there is a real gain.

⋆ This work is supported by project TACOS ANR-06-SETI-017 and by project CRISTAL

[1] http://www.fsel.com/

[2] http://www.clearsy.com

2 Requirements for the tool

2.1 CSP||B and verification

CSP||B is a multi-formalisms specification technique developed in the last 10 years. It focuses on the modeling of component-based systems. A component is an entity which manages some internal state and which communicates with other components. B is used to describe the state management part of the component. CSP is used to describe the communication part of the system.

A component is modeled by two texts: a B machine and a CSP controler. To each operation of the B machine is associated a "machine channel" in the CSP controler. The semantics is that when a process synchronizes on a machine channel, this corresponds to a " call" of the machine operation.

The theory of the verification of a CSP||B specification has been developed in [1]. It is based on a process with three independent steps:

- the B machines are verified by the usual B technology: generation of proof obligations and proofs with B provers,
- the CSP controlers are verified by standard CSP techniques to assess the absence of deadlocks or divergences,
- the protocol constraints expressed in CSP processes guarantee that the precondition of an operation holds when it is called through a machine channel synchronization.

This last step is a consequence of two implicit assumptions in the preceding steps: B machine invariants are maintained only if operations are called when their precondition holds and CSP synchronizations always happen when they are possible, which is always the case for "calling" a machine operation through a machine channel synchronization. Hence, CSP||B mandates that operations in machines are coded with preconditions instead of guards since the B semantics states that the former are always executable but may lead to any result if the preconditions do not hold.

2.2 Expected functions

[1] proposes a verification technique where CSP controlers are translated into B machines. The calling order imposed by the protocol is modeled by a counter. The translation involves two kinds of works. The first is the application of a set of transformation rules, it is the easy but tedious part. The second is the introduction of a *Control Loop Invariant (CLI)*, which is the intelligent part. CLI is such that

$$CLI = (cb = 1 \implies PRE(op_1)) \wedge \ldots \wedge (cb = n \implies PRE(op_n))$$

where $PRE(op_i)$ is the precondition of the operation called when $cb = i$. CLI can be simplified by removing some conjuncts after analysis of the structure of the operations.

The first and most important requirement is to apply the transformation rules and generate files that can be used by AtelierB. The second requirement is to assist specifiers with invention of the CLI. Several strategies have been studied, from making CLI a parameter of the translation program to generating it. We ended up with the idea to generate the form of the CLI needed by the specification.

3 Realization

3.1 Architecture and design

We see our tool as a protoytpe and, thus, followed a process based on short and fast cycles. Architecture and design are, in a sense, results of the development.

An important decision was the structure of the output of the translation. For each CSP controler it consists of two controler machines: an abstract machine which introduces the counter and a refinement which introduces the calls to the operations of the controled B machine. When needed, CLI are introduced in the refinements.

The current translation process is a sequence of six semantic steps which follow the parsing of the files containing the CSP controler and the B machine:

1. generation of the abstract machine, as a text file,
2. removal of non pertinent variables. CSP syntax requires to introduce names when values are passed through a channel. With respect to the verification of coherence between B and CSP, only the names associated with machine channel are pertinent and must be translated, all others can be safely ignored,
3. removing name conflict. A pertinent variable can lead to a conflict on three cases: it is one letter long, it is identical to a keyword of B, it is identical to an existing B variable visible through the `include` clause,
4. type inference. B, contrary to CSP, requires all variables to be typed,
5. computation of the CLI form,
6. generation of the refinement, as a text file.

3.2 Implementation and test

As an implementation language, we used OCaml[3]. Three reasons motivated this choice: OCaml is a powerful language for symbolic manipulations, it has strong parsing libraries, and the project Brillant[4] provides us with several tools, notably a B parser.

The implementation consists in two packages: one for B analysis, around the Brillant parser, and one for CSP analysis. A parser for CSP, accepting the syntax used by FDR2, has been implemented.

The tool needs two input files: `C.mch` and `C.csp` which contain the texts for the B part and the CSP controler. Included files in CSP texts are read when needed.

A required CLI is indicated by a comment containing the skeleton of the invariant.

The program has been tested on three published specifications [2,3,4]. It was used to verify the extension of [4] which motivated this development.

4 Observations

Picking OCaml for developing the tool has been an excellent choice. Beside the Brillant library, the major advantage is the clarity of the code. Ideas could be implemented very

[3] `http://caml.inria.fr`
[4] `https://gna.org/projects/brillant`

quickly and the intuitive correctness of the algorithms is easy to assess. In the three months of programming, we were able to test several translation strategies.

Starting without a firm design is another advantage of fast programming. In fact, we "discovered" the technical problems (typing, name clashes, etc.) as we progressed and processed new exemples. Of course, production tools must use much more structured development processes. However, such processes rely on that, from the outset, the shape of the solution is well known. Our work only aims at drawing the shape!

A problem that we encountered is the paucity of exemples of specifications. This limited the territory which our exploratory approach could chart. While we have a reasonable confidence in the correctness of our prototype, we do not know how many difficulties, either technical or theoretical, remain to discover. For instance, we know that the generated CLI is pessimistic when operations contain a conditional substitution. Which other traits of the languages could cause a similar phenomenon should be studied. A library of exemples would be a real bonus here.

Another problem is the communication between the tools which should enter the toolbox of a specifier. In our case, we used the minimum common denominator: text files. Inefficiency is not only technical, repeatedly parsing files, it is also conceptual. Parsers are complicated to build and to use, and thus consumme a lot of developer's time, yet the interesting points are elsewhere. The problems cited above could be better solved if we had libraries with "semantic" functions such as the set of variables modified by an operation, or read only, etc. Such libraries are much needed.

5 Conclusion

A question remains about the potential utility of a tool such as ours: do we need it in the toolbox? Should we invest in the development of a "certified" and stronger version, knowing that it does an incomplete job? We think so. Even for a formal developer, not all tools need be fully formal and complete. Time is short, boring tasks lead to errors. For the practical developer, there is room in the box for tools which take care of mundaneness and leave out the smart bits.

Our prototype is available to brave users through the Brillant project.

References

1. Treharne, H.: Combining Control Executives and Software Specifications. PhD thesis, University of London (2000)
2. Schneider, S., Treharne, H.: Communicating B machines. In Bert, D., Bowen, J.P., Henson, M.C., Robinson, K., eds.: Formal specification and development in Z and B (ZB 2002). Volume 2272 of LNCS., Springer Verlag (2002) 416–435
3. Treharne, H.E., Schneider, S.A., Bramble, M.: Combining specification with composition. In: ZB2003: International Conference of Z and B Users. Volume 2651 of LNCS., Springer (2003)
4. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Towards validating a platoon of cristal vehicles using CSP||B. In Springer, ed.: 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008), France (2008-07) 6 pages