A Support Tool for CSP_Z Data Abstraction

Adalberto Farias, Alexandre Mota, and Augusto Sampaio

Universidade Federal de Pernambuco, Centro de Informática,
Av. Prof. Luis Freire, s/n, Cidade Universitária, Recife,
Pernambuco, Brazil. CEP 50740-540

Abstract. Although model checking is an automatic technique, it is not able to analyse infinite state systems due to the state explosion problem. Compression techniques have been employed to avoid such a problem. A very successful technique is data abstraction which replaces original (infinite) data domains with new (finite) ones while still preserving the properties of the original system. In this work, we show a flexible tool for analysing infinite CSP_Z processes based on data abstraction.
Keywords: CSP_Z, specification, model checking, mechanical data abstraction, support tool.

1 Introduction

Integrated languages have been employed in concurrent system modelling in the last years. With those languages, it is possible to model different aspects (such as behaviour, data, time, mobility) at once. In general, process algebras like CSP [10,17] or CCS [13] are more adequate to model behavioural aspects (processes and interactions among them), whereas model-based languages like Z [19] or VDM [12] are more adequate to describe data aspects. The integration of such formalisms originates a new one, whose expressiveness permits to model such aspects. For instance, CSP_Z is a formal combination of CSP and Z in such a way that one can model both behavioural (the CSP part) and data (the Z part) aspects simultaneously and orthogonally.

After specifying a system, tool support is essential to find desirable properties of it. Normally, the technique employed to analyse systems is model checking [9]. Although this technique is completely automatic, it is limited to finite-state systems. This has motivated the community to investigate many compression techniques [1,14,17]: elimination of symmetry, abstraction, symbolic execution, data independence and integration of tools (like, for example, model checkers and theorem provers).

The first mechanised strategy for analysing infinite CSP_Z processes was proposed in [14,16], where an infinite system is transformed into a finite one by replacing the original data type (concrete) with an abstract version.

The strategy presented in [2,3] is an improved extension of that presented in [14,16], such that the control (CSP) part is also considered when abstracting a CSP_Z process. Therefore, special situations only manifested when considering the behavioural part are correctly captured by that approach and the strategy becomes more efficient. Furthermore, a tool supporting the strategy was also presented.

In this work we present a tool which implements the idea of abstracting a CSP_Z process by considering its two parts: control and data. As theorem proving can be required while applying the strategy, the tool also interacts with theorem provers. The use of patterns [7] during the implementation allows one to supply extra modules in order to interact with other theorem provers (the default theorem prover used by the tool is Z-Eves [18]). Details about the theory to achieve the strategy and the algorithm can be found in [2].

The following sections present an overview of CSP_Z, the strategy for abstracting CSP_Z processes, the support tool and our conclusions about this work.

2 The CSP_Z Language

The specification language CSP_Z is a formal integration of CSP and Z, such that CSP is used to describe behaviour and Z is used to describe data structures, state-space and operations. Its formal semantics has been defined in terms of CSP [4,5]. But informally we can describe a CSP_Z specification as a parallel combination of two complementary parts where the CSP part (P_{CSP}) acts as a master process and the Z one (P_Z) achieves all data manipulation. According to the strategy defined in [15], a CSP_Z specification is translated into pure CSP so that it can be analysed with the FDR tool [6].

Example 1. This specification describes a simple clock which infinitely performs two events.

spec $Clock$

$chan\ tick,\ tack$	(channel declaration – CSP part)
$main = tick \rightarrow tack \rightarrow main$	(main process – CSP part)
$State \mathrel{\widehat=} [c : \mathbb{N}]$	(state definition – Z part)
$Init \mathrel{\widehat=} [State' \mid c' = 0]$	(initialisation – Z part)
$com\_tick \mathrel{\widehat=} [\Delta State \mid c' = c + 1]$	(operation – Z part)
$com\_tack \mathrel{\widehat=} [\Delta State \mid c' = c + 1]$	(operation – Z part)

end_spec P

In the above example, we have two channels, *tick* and *tack*. The process *main* (represents the CSP part) infinitely performs the sequence of events $\langle tick, tack \rangle$. On the Z part, schemas *State*, *Init*, *com_tick* and *com_tack* define the state, initialisation and operations respectively. The state contains a variable representing an internal counter (c), whose value is a natural number ($\mathbb{N}$). The initialisation assigns 0 to the state variable, and the operations, which are always enabled (preconditions *true* are omitted), change the state by incrementing the value of c ($c' = c + 1$). While the CSP part performs *tick*, the Z part executes the corresponding schema *com_tick* and so on. Figure 1.a gives a view in terms of a labelled transition system (LTS).

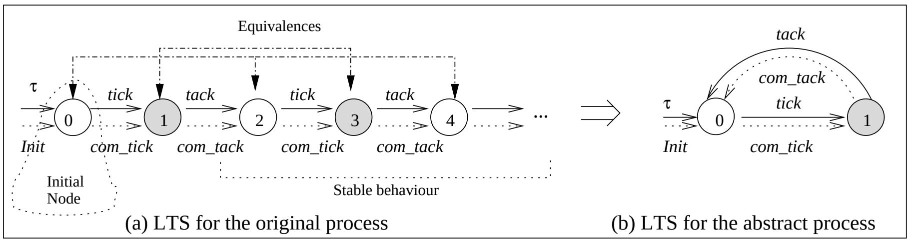

Fig. 1. The abstraction of *Clock*

3 Data Abstraction

The data abstraction technique is a compression technique which allows one to overcome the state explosion problem. In the context of CSP_Z, the idea is deriving another specification by changing infinite types to finite ones, while still preserving the behaviour of the whole process. The abstraction is obtained by identifying subsets to represent the original types, through a mathematical partitioning. Broadly, the algorithm identifies a class of values (possibly an infinite partition) of the original type with a single value that represents the entire class (partition). This is achieved by showing that the result produced by symbolic execution of the process under analysis is the same for every value in the partition. Applying the algorithm proposed in [2] to Example 1, we derive:

spec $AbsClock$

$chan\ tick,\ tack$

$main = tick \rightarrow tack \rightarrow main$

$\mathcal{A} = \{0, 1\}$

$State \mathrel{\widehat=} [c : \mathcal{A}]$ $\qquad$ $com\_tick \mathrel{\widehat=} [\Delta State \mid c' = 1]$

$Init \mathrel{\widehat=} [State' \mid c' = 0]$ $\qquad$ $com\_tack \mathrel{\widehat=} [\Delta State \mid c' = 0]$

end_spec $AbsClock$

Note that the above specification is a CSP_Z process which performs the same events as the original one, but can be finitely represented (see Figure 1.b).

4 Tool Support

The tool achieving the CSP_Z data abstraction approach has been developed in Java [11], and provides support both to translating a specification and to abstracting a CSP_Z process by applying the algorithm proposed in [2]. The main modules of the tool (see Figure 2) are: *Parser*, *Data Independence*, *Data Abstraction* and *Translator*.

The *Parser* is composed of two internal parsers: one for CSP and another for Z. It accepts an input file where a CSP_Z specification is written, and generates a syntactical tree. The *Data Independence* module analyses such a syntactical in order to assure that the CSP part is trivially data independent, that is, it works similarly with any data type. The *Data Abstraction* module applies the technique of abstracting a process without causing semantic loss (the abstracted process performs the same events as the original one). The *Translator* module accepts a syntactical tree of a CSP_Z process and generates a CSP specification to be analysed into FDR.

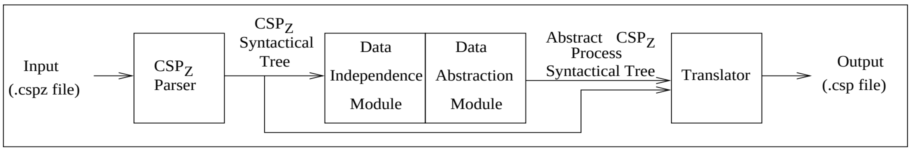

Fig. 2. The modules of the tool

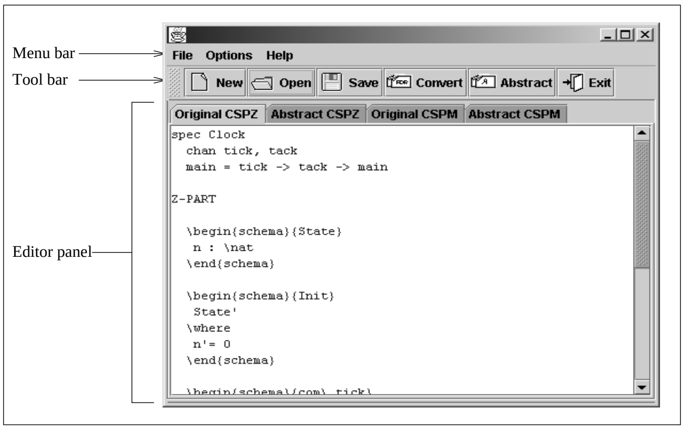

Fig. 3. Main screen of the tool

The tool also has support for editing the files (see Figure 3) and is available at `http://www.cin.ufpe.br/~acf`.

5 Conclusions

In this work we have reported a tool for giving support to CSP_Z data abstraction, allowing one to model check infinite systems. The tool is extensible, in the sense that the user can provide modules to interact with different theorem provers and model checkers.

A similar work is reported in [8], which proposes another compression technique based on predicate abstraction. That work has been used in the SLAM project (details at `http://research.microsoft.com/slam`).

The literature does not furnish any approach dealing with mechanised abstractions involving communication to the best of our knowledge. Thus, for future research, we intend to extend our strategy towards this direction and incorporate such a feature into our tool in order to turn it able to deal with more complex CSP_Z specifications.

References

1. Cleaveland, R. and Riely, J. Testing-based abstractions for value-passing systems. In J. P. B. Jonsson, editor, *CONCUR'94*, volume 836, pages 417–432. Springer-Verlag, Berlin, 1994.
2. Farias, A. Efficient and Mechanised Analysis of Infinite CSP_Z Processes: strategy and tool support. M.Sc. dissertation, 2003.
3. A. Farias, A. Mota and A. Sampaio. Efficient Analysis of Infinite CSP_Z Processes. *In Proceedings of V Workshop on formal Methods* (*WMF2002*), pages 113 – 128.
4. Fischer, C. Combining CSP and Z. Technical Report, University of Oldenburg, 1996.
5. Fischer, C. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.
6. Formal Systems (Europe). FDR2 User Manual, 1997.
7. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1998.
8. Graf, S. and Saïdi, H. Construction of abstract state graphs with PVS. In *CAV 97: Computer-Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
9. Grumberg, O., Clarke, E. and Peled, D. *Model Checking*. The MIT Press, Cambrige, MA, 1999.
10. Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.
11. Horstman, C. and Cornell, G. *Core Java 2*. Sun Microsystems Press, volumes I and II, 2000.
12. ISO. Information technology - Programming languages, their environments and system software interfaces - *Vienna Development Method - Specification Language* - Part 1: Base language. International Standard ISO/IEC 13817-1, December 1996.
13. Milner, R. In *A Calculus of Communicating Systems,* Lecture Notes in Computer Science, vol. 92, Springer-Verlag, Berlin, 1980.
14. Mota, A. *Model Checking CSP_Z: Techniques to Overcome State Explosion*. PhD thesis. Federal University of Pernambuco, Brazil, 2002.
15. Mota, A. and Sampaio, A. Model-Checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, Elsevier, Netherlands. **(40)1**:59–96, 2001.
16. Mota, A., Sampaio, A. and Borba, P. *Mechanical Abstraction of CSP_Z Processes*. FME 2002, LNCS 2391, pp. 163–183.
17. Roscoe, A. W. *The Theory and Practice of Concurrency*. Oxford University, 1998.
18. Saaltink, M. Z-Eves System. In ZUM'97: the Formal Specification Notation, volume 1212, LNCS, Springer, 1992.
19. Spivey, M. *The Z Notation: A Reference Manual, 2nd Edition*. Prentice Hall International, Englewood Cliffs, NJ, 1992.