Z2SAL: a translation-based model checker for Z

John Derrick, Siobhán North and Anthony J. H. Simons
Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK.
E-mail: J.Derrick@dcs.shef.ac.uk

Abstract. Despite being widely known and accepted in industry, the Z formal specification language has not so far been well supported by automated verification tools, mostly because of the challenges in handling the abstraction of the language. In this paper we discuss a novel approach to building a model-checker for Z, which involves implementing a translation from Z into SAL, the input language for the Symbolic Analysis Laboratory, a toolset which includes a number of model-checkers and a simulator. The Z2SAL translation deals with a number of important issues, including: mapping unbounded, abstract specifications into bounded, finite models amenable to a BDD-based symbolic checker; converting a non-constructive and piecemeal style of functional specification into a deterministic, automaton-based style of specification; and supporting the rich set-based vocabulary of the Z mathematical toolkit. This paper discusses progress made towards implementing as complete and faithful a translation as possible, while highlighting certain assumptions, respecting certain limitations and making use of available optimisations. The translation is illustrated throughout with examples; and a complete working example is presented, together with performance data.

Keywords: Z, model-checking, SAL

1. Introduction

Despite being widely known and accepted by the software industry, the formal notation Z [Spi92] has for some time lagged behind other specification languages in the provision of tools for automatically verifying specifications, whether by simulation, model-checking or theorem proving. There are a number of reasons for this, although most are connected with the language itself and its semantics: the inherent expressivity of Z makes it harder to build tractable tools for it.

Historically, early tools, such as fuZZ [Spi00] and CADiZ [TM95], were closely linked to the typesetting languages, troff and LATEX, used to write Z and focused on creating, formatting and type-checking Z specifications. Later versions of CADiZ and the Z/Eves composer and proof tool [Saa97, Saa99] were able to perform additional functions, such as domain checking (stricter than type checking, for partial functions), schema expansion, with redundant term elimination, and interactive theorem-proving using heuristics and proof tactics suggested by the user. CADiZ is under continuous development and has since evolved towards the ISO-Z standard.
1.1. Community Z tools

More recently, a concerted effort has been made in the wider global Z user community to address the general tool deficiency. The Community Z Tools (CZT) project [MFMU05] is one leading example. This group is in the process of developing a set of open source tools for Z, based around the ZML markup language [DUT’03], an XML dialect developed specifically for Standard ISO-Z [135]. So far, there is a parser and a type-checker for Z, an AST package developed in Java for use in third-party modules, and a number of other proposed modules, including cross-language translators and model-checkers, based on the same parser and AST. The tools handle a number of input formats, including ZML and \LaTeX. This work is foundational and will provide long term grass-roots support for Z. However, the progress towards finished provers and checkers is slow, partly due to the complexity of the Z standard and the use of automatic code generation technology to develop the parser and AST, whose API needs more user-friendly documentation, to encourage a wider take-up.

Elsewhere, others have sought a quicker route to developing model-checkers for Z by adapting existing tools that support automated checking. An example of this is the ProZ tool [PL07], which extends and adapts the earlier ProB tool [LB05] for Z. The B language [Abr96], though related to Z, is much closer to the state-transition formalism used by symbolic model-checkers. ProB and ProZ both use an underlying Prolog engine to simulate (or “animate”) a specification, by populating variable terms with ground values, chosen from restricted ranges. Configurations of state variables form the states of the automaton, while each operation is styled as a transition from state to state, affecting the values of variables. The validity of models is checked by simulating forwards in time from the initial state, exploring multiple future states in parallel. Consistency is checked by verifying the state invariant in each state, or by detecting deadlocks (failure to instantiate all variable terms). A further facility exists for checking refinement relations between specifications.

A similar approach was taken by Bolton [Bol05], who used the Alloy SAT-solver based counter-example finder [Jac02] to verify data refinements in Z, after translating from Z into the Alloy input language. This is similar to our philosophy [DNS06] of translating Z into the input language for the SAL tool-suite [dMOS03], which uses a BDD-based symbolic model checker as its core engine. We consider that this strategy of translating into the input format of another proven toolset will result in better model-checking capabilities than building a bespoke model-checker for Z on top of the CZT toolkit. The SAL core engine is already quite sophisticated, transforming set-theoretic and mathematical formulae into boolean judgements on ordered variables, which are compiled to optimized binary decision trees (BDDs), before generating Büchi automata for each theorem expressed in temporal logic to explore the compacted state space. Developing a similar tool from scratch would require considerably more effort than building a cross-translator.

1.2. The symbolic analysis laboratory

Our choice of the symbolic analysis laboratory (SAL) tool-suite [dMOS03] as the target for the translation was motivated by a number of reasons. Firstly, there were already a number of different tools using the SAL input language. These included a simulator, a model-checker, a bounded model-checker and a counterexample finder, with other tools in the pipeline. Secondly, there was a sizeable international user-group engaged in developing and using the tools, offered gratis by SRI under an academic licence, with some support offered from the developers. Thirdly, the SAL input language [dMOS03] is purposely designed to be formalism-neutral, positioning itself somewhere between the highly restrictive machine-centric syntax required by Spin [Hol97] and SMV [CGL94], and the complete expressiveness offered by conventional programming languages. The SAL input language supports finite sets, tuples, subranges, arrays, records, total functions, concurrently executing and parameterised modules and (in principle) recursive definitions. The kinds of theorem that may be checked include first-order predicate terms and both LTL and CTL temporal logic expressions. Finally, the core SAL engine compiles all definitions into optimized binary decision trees (BDDs) and simulates models using Büchi automata, proven approaches for dealing with large state spaces efficiently.

The original idea of translating Z into SAL specifications was due to Smith and Wildman [SW05]. In [DNS06] we described the basics of our implementation, which is essentially a bespoke parser and generator, written in Java, which translates from the \LaTeX encoding of Z into the SAL input language. The collection of Z schemas is translated into a SAL finite state automaton, following a template-driven strategy with a number of associated heuristics. Like [SW05], we aim to preserve the Z-style of specification, including postconditions that mix primed