Building models through formal specification

Gerrit Renker, Hatem Ahriz

School of Computing
The Robert Gordon University, Aberdeen. Scotland, UK.
{gr,ha}@comp.rgu.ac.uk

Abstract. Over the past years, a number of increasingly expressive languages for modelling constraint and optimisation problems have evolved. In developing a strategy to ease the complexity of building models for constraint and optimisation problems, we have asked ourselves whether, for modelling purposes, it is really necessary to introduce more new languages and notations. We have analyzed several emerging languages and formal notations and found (to our surprise) that the already existing Z notation, although not previously used in this context, proves to a high degree expressive, adaptable, and useful for the construction of problem models. To substantiate these claims, we have both compiled a large number of constraint and optimisation problems as formal Z specifications and translated models from a variety of constraint languages into Z. The results are available as an online library of model specifications, which we make openly available to the modelling community.

1 Motivation

Formal methods and notations are most commonly associated with software development in procedural and object-oriented implementation languages. We are developing a strategic software engineering approach for modelling constraint and optimisation problems (csops); one of the main underlying objectives is to integrate the notion of such problems into the standard software design cycle [8]. For this purpose, we have been investigating the use of formal notation in general and of Z in particular, coming to the conclusion that advantages are to be had in at least four areas.

The first concerns the inception phase of building an initial or conceptual model. A modeller must first come up with an understanding of the problem requirements before being able to exploit its specific features. Quoting Smith, a recognized expert in the area of modelling: "*Hence, although constraint programming does require an understanding of search and constraint propagation, it is by understanding the problem and building in that understanding that we can develop a successful model.*" [9, sec. 13]

Secondly, as larger-scale software is mostly developed in a (possibly distributed) team context and problem-solving strategies are shared across the modelling community, we see the importance of formal notation as a means of communication which is not constrained by and tied to the specifics of a particular implementation language. Specifications of constraint and optimisation problems in the scientific literature are often either based on the use of non-standardized (sometimes informal) mathematical notation, or in form of source-code descriptions at implementation level. The importance of not committing to a certain implementation format was also one of the crucial insights realized by the founders of CSPLib,[1] as "*representation remains a major issue in the success or failure to solve constraint satisfaction problems. All problems are therefore specified in natural language* ..." [3]

The third aspect lies in proving and verifying that a constraint problem at hand is indeed syntactically and semantically covered by a given model. Formal specification languages here allow the interaction with computer tools (figure 1) for simplifying, reducing and rewriting statements, thus allowing to generate (canonical) forms of expressions which are either more general or more suitable for the problem at hand. The notion of debugging in constraint programming fundamentally differs from that in procedural programming, a verified model specification reduces the need for debugging by highlighting conceptual errors at an early stage of development. We support the argumentation of Law and Lee in [6] in that we would like to reason about properties of CSP models without actually having to solve these. Modelling in constraint programming is further, like mathematical modelling, a rather abstract mental activity, and so the verification[2] of a model can provide concrete evidence, reassuring the modeller that a chosen concept is indeed correct.

Finally, and in keeping with evolving concepts of constraint problem modelling, formal specifications allow higher-level abstractions of model formulations. Over the past years increasingly more expressive and abstract modelling languages have evolved. OPL [4] innovated a uniform abstraction to deal with both CP and OR problems at the same language level. The $\mathcal{F}$ language [5] introduced useful model abstractions based on function variables, which is being developed further in the ESRA language for relational modelling of constraint problems [2]. Work on automated model refinement [1] has provided substantial support for the conjecture that constraint problem models can be constructed by compositional refinement of abstract specifications. Such compositionality is also at the heart of introducing algebraic CSP model operators to support a modular design of constraint problem formulations [6]. Furthermore, Law and Lee speak in that study of "*reusable model components*" and "*model patterns*" [6, sec. 5]. The latter recently stirred interest in form of an invited lecture [12].

From the above considerations we have chosen Z [10,11], due to the fact that its style is generic and not geared towards a certain programming paradigm. The schema format, as introduced in section 2, proved a natural match for expressing the main bodies of constraint and optimisation problems. To substantiate our claims and to evaluate Z, we have compiled a large number of well-known constraint and optimisation problems, which we make openly available as online library of specifications to the modelling community (cf. section 5). The remain-

[1] `http://www.csplib.org`

[2] A formal specification can also prototypically be verified through animation.

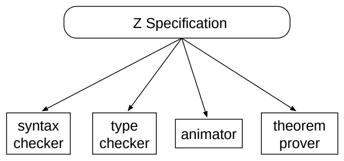

Fig. 1. Further processing of model specifications

der of this document is structured as follows. After a brief summary of relevant Z features in section 2, we show how to use Z for the specification of CSOPs in section 3, followed by an example in section 4 and conclusions in section 5.

2 A brief recapitulation of Z features

Z is a typed formal specification language based on first-order logic and Zermelo-Fraenkel set theory. It provides a precise syntax and a semantics based on classical mathematics for the abstract specification of systems in a model-oriented way. The language has been standardized as ISO/IEC standard 13568:2002, and its reference manual [11] comes with a mathematical toolkit of common operations on sets and numbers. Main elements of a Z specification are given sets, axiomatic definitions and schemas. *Given sets* are introduced as further unspecified global names within square brackets, e.g.

$[\mathit{Warehouses}]$

This allows to reference the set *Warehouses* as type throughout the specification. *Axiomatic definitions* also have global scope and are often used to introduce constants or constant mappings. An axiomatic definition consists of a declaration part and an optional predicate part, separated by a horizontal line.

$$\begin{array}{|l} \mathit{square} : \mathbb{N} \longrightarrow \mathbb{N} \\ \hline \forall\, n : \mathbb{N} \bullet \mathit{square}(n) = n * n \end{array}$$

The example[3] introduces a total function *square* on $\mathbb{N}$. Several type constructors, e.g. tuples, Cartesian product and (finite) power-sets, are provided by default, as well as common mathematical data types such as relations, functions, sequences and bags. Composite and heterogeneous data types can be introduced using schemas, which are one of the most powerful features of Z. A *schema* is an elementary building block of a Z specification. Like axiomatic definitions, schemas divide into a declaration and optional predicate part, the difference being that all declared constants and variables are locally-scoped. For example,

[3] this example first appeared in [10, pp. 123/24].

SQPAIR

$x, y : \mathbb{N}$

$y = square(x)$

Here, x, y are local to *SQPAIR* and y is assigned the value of applying the global function *square*. The elements in the declaration part are called *components* of the schema. A schema can therefore be viewed as a set of named components that are constrained by predicates. Schemas can be combined into new ones using the operations of the schema calculus such as inclusion, composition, projection, conjunction, disjunction, negation and hiding. A schema can also be seen as a mere abbreviation for the text it contains. Instead of

$$\exists\, x, y : \mathbb{N} \bullet y = square(x) \land x > 100$$

we can equivalently write $\exists\, SQPAIR \bullet x > 100$. The *type* of a schema is the signature of its components, where the order of appearances is irrelevant. The type of the above schema is $\langle\!|\, x : \mathbb{N};\, y : \mathbb{N} \,|\!\rangle$. Likewise, the term $\{SQPAIR\}$ is the set of all schema bindings which have the type $\langle\!|\, x, y : \mathbb{N} \,|\!\rangle$ and contain exactly those values for x, y such that $y = square(x)$. More sophisticated variants of schemas in Z allow generic and parameterised definitions that specify entire families of schemas rather than sets of complying objects [11].

3 Adapting Z for constraint and optimisation problems

Constraint satisfaction problems are usually defined as a triple $\langle X, D, C\rangle$ of variables X, domains D and constraints C formulated over X. In the majority of constraint (logic) programming languages, the constraints in C can be expressed as quantified formulae of first-order logic. This allows a representation of constraint satisfaction problems in Z by single schemata, named e.g. *CSP*,[4] where the elements of X and D are contained in the declaration part and the constraints in C in the predicate part. In cases of complex domains d_i, the base type (e.g. $\mathbb{Z}$) appears in the declaration part in combination with additional unary constraints on x_i in the predicate block. These concepts are illustrated by the example in section 4. Following the semantics of Z [10], the solution set of constraint problems defined in the aforementioned way is simply the set $\{CSP\}$ (wrt. the above schema name), since it is the set containing all objects of type $\langle\!|\, x_1 : d_1 \ldots, x_n : d_n \,|\!\rangle$ such that the constraints C of the predicate block hold. This permits the definition of a template for specifying optimisation problems. In *constrained optimisation problems*, we are interested in selecting the 'best' out of a set of solutions to a problem, where the evaluation criterion is determined by an objective function mapping solutions into numerical values. As is customary in IP and many CP languages, we will here assume that objective functions range over $\mathbb{Z}$. Using the above format for expressing constraint satisfaction problems, let the constraints of the problem be given as a schema *CSOP*.

[4] Besides, we can also make modular use of other and auxiliary schemata.

We can then define the objective function in a separate schema, as a function from $CSOP$ to $\mathbb{Z}$, and express the solution of the problem in terms of optimising the value of this function. This is also illustrated in section 4. The procedure for *unconstrained optimisation problems* is the same as for the constrained variants, the difference being that the predicate block of the main constraint schema remains empty. As Z itself does not make restrictions on the domains to use, we can in principle extend the concept also to the domain of Real (or even complex) numbers, although we would need to supply an appropriate toolkit.

4 An example specification

We now illustrate the main concepts of the last section on a small example,[5] the bus driver scheduling problem (`prob022` in CSPLib). We are a given set of tasks (pieces of work) and a set of shifts, each covering a subset of the tasks and each with an associated, uniform cost. The shifts need to be partitioned such that each task is covered exactly once, the objective is to minimise the total number of shifts required to cover the tasks. The sets of interest are *pieces* and *shifts*,

$$\mid\ pieces, shifts : \mathbb{P}\,\mathbb{N}$$

defined here as sets of natural numbers. The function *coverage* is part of the instance data and denotes the possible subsets of *pieces*. The only decision

$$\begin{array}{|l}
\textit{Driver\_Schedule} \\
\hline
coverage : shifts \longrightarrow \mathbb{F}\, pieces \\
allocate : \mathrm{iseq}\, shifts \\
\hline
coverage \circ allocate \ \mathsf{partition}\ pieces \\
\hline
\end{array}$$

variable is *allocate*, an injective[6] sequence of *shifts*. Composition of *allocate* with *coverage* yields a sequence of subsets of *pieces*. The built-in partition operator of Z [11, p. 122] asserts that this sequence of subsets is a partition of the set *pieces*. We continue with the optimisation part, which illustrates the template format for modelling optimisation problems we mentioned in section 3. The *objective* function maps each element from the solution set *Driver_Schedule* into a natural number, in this case the number of shifts represented as the cardinality of the *allocate* variable.

$$\begin{array}{|l}
\textit{Optimisation\_Part} \\
\hline
objective : Driver\_Schedule \longrightarrow \mathbb{N} \\
solution : Driver\_Schedule \\
\hline
\forall\, ds : Driver\_Schedule \bullet objective(ds) = \#(ds.allocate) \\
objective(solution) = \min(objective(\!| Driver\_Schedule |\!)) \\
\hline
\end{array}$$

[5] as one reviewer rightly pointed out, the set-based nature of this small example is not very indicative of Z's abstraction facilities, but this example shows how succinct a formulation is possible. We refer to the more than 50 online examples.

[6] shifts can appear at most once.

The last expression states that the element *solution* of the solution set must have a minimal value of the *objective* function. For this purpose, we use relational image of the entire solution set *Driver_Schedule* through *objective*.

5 Conclusion and further work

In this paper we have summarized the successful use of Z as a precise modelling notation for CSOPs. With regards to expressivity, we had positive results in mapping constructs and models from OPL, ESRA and $\mathcal{F}$ [7]. We initially wrote the specifications without any tool support, subsequent verification (using the *f*uzz type checker[7] and the Z-Eves[8] prover) however proved so helpful in ironing out inconsistencies and improving the understanding of the problems that now all models are electronically verified prior to documentation. Our main focus at the moment is the modelling strategy and formal analysis of models. Aspects of further investigation are the translation of Z models into an implementation language, model animation and further tool support. The online repository is at `http://www.comp.rgu.ac.uk/staff/gr/ZCSP/`.

Acknowledgment. We kindly thank the reviewers for their constructive criticism.

References

1. A. Bakewell, A. M. Frisch, and I. Miguel. Towards Automatic Modelling of Constraint Satisfaction Problems: A System Based on Compositional Refinement. In *Proceedings of the Reform-03 workshop, co-located with CP-03*, pages 2–17, 2003.
2. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a Relational Language for Modelling Combinatorial Problems. In *Proc. Reform-03*, pages 63–77, 2003.
3. I. P. Gent and T. Walsh. CSPLib: A Benchmark Library for Constraints. In J. Jaffar, editor, *Proceedings of CP'99*, pages 480–481. Springer, 1999.
4. P. V. Hentenryck. *The OPL Optimization Programming Language.* MIT, 1999.
5. B. Hnich. *Function Variables for Constraint Programming.* PhD thesis, Department of Information Science, Uppsala University, Sweden, 2003.
6. Y. C. Law and J. H. M. Lee. Algebraic Properties of CSP Model Operators. In *Proceedings of Reform-02, co-located with CP'02*, pages pp. 57–71, 2002.
7. G. Renker. A comparison between the F language and the Z notation. Technical report, Constraints Group, Robert Gordon University, Aberdeen, November 2003.
8. G. Renker, H. Ahriz, and I. Arana. A Synergy of Modelling for Constraint Problems. In *Proc. KES'03*, volume 2773 of *LNAI*, pages 1030–1038. Springer, 2003.
9. B. Smith. Constraint Programming in Practice: Scheduling a Rehearsal. Technical Report APES-67-2003, APES Research Group, September 2003.
10. J. M. Spivey. *Understanding Z: A specification language and its formal semantics*, volume 3 of *Cambridge tracts in theoretical computer science.* CUP, 1988.
11. J. M. Spivey. *The Z Notation: A Reference Manual.* Oriel College, Oxford, 1998.
12. T. Walsh. Constraint Patterns. In *Proc. CP'03*, pages 53–64. Springer, 2003.

[7] `http://spivey.oriel.ox.ac.uk/mike/fuzz/`
[8] `http://www.ora.on.ca/z-eves/`