From Software Specifications to Constraint Programming

Stefan Hallerstede[1], Miran Hasanagić[1], Sebastian Krings[2], Peter Gorm Larsen[1] and Michael Leuschel[2]

[1] Department of Engineering, Aarhus University, Denmark
[2] University of Düsseldorf, Düsseldorf, Germany

Abstract. Non-deterministic specifications play a central role in the use of formal methods for software development. Such specifications can be more readable, but hard to execute efficiently due to the usually large search space. Constraint programming offers advanced algorithms and heuristics for solving certain non-deterministic models. Unfortunately, this requires writing models in a form suitable for efficient solving where the readability typically required from a specification is lost. Tools like ProB attempt to bridge this gap by translating high-level first-order predicate logic specifications into formal models suitable for constraint solving. In this paper we study potential improvements to this methodology by (1) using refinement to transform specifications into models suitable for efficient solving, (2) translating first-order predicates directly into the OscaR framework and (3) using different kinds of solvers as a back end. Formal verification by proof ensures the correctness of the solution of the model with respect to the specification.

1 Introduction

State-based modelling methods like B and VDM support writing abstract specifications and implement them by refinement. Refinement is carried out semi-automatically, leading to deterministic implementations which are amenable to automatic code generation. A long term ambition of software tools like ProB [17] is to allow users to write very high-level specifications, which are easy to read and write, amenable to formal proof and can yet still be executed efficiently. In other words, specifications are viewed as non-deterministic programs, where constraint solving is used to compute solutions to their formal specifications [18]. Still, the efficient solution of hard constraint satisfaction problems often requires an encoding of formal descriptions of the problems that is difficult to comprehend. Such concerns turn specifications into non-deterministic implementations and open up the possibility for programming errors just as in the case of deterministic implementations. As a consequence, we benefit less from the declarative paradigm than we would expect. In this paper we propose to use formal refinement as a method to relate formal models *SPEC* that describe problems at a high level of abstraction to formal models *AIMP* at a lower level of abstraction targeting constraint solvers. In the same way as for the deterministic

case, refinement achieves correctness with respect to an easier to understand abstract specification. Figure 1 illustrates the approach. The low-level models

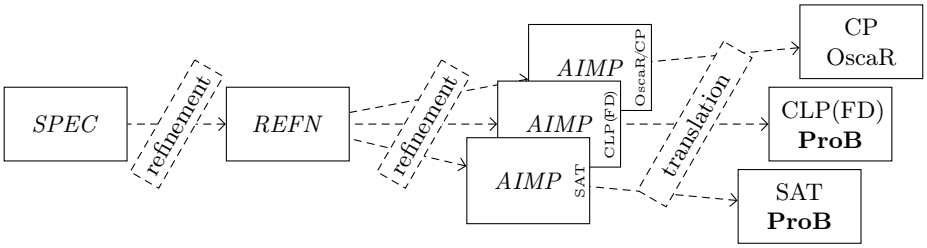

Fig. 1. Refinement and translation of abstract models

can be translated into different frameworks like Constraint Programming (CP), Constraint Logic Programming over Finite Domains (CLP(FD)) or SAT for efficient execution according to their "flavour". The different models may still share modelling concepts but use specific concepts with efficient representations in the target frameworks. In order to ease formal refinement proofs, typically several refinement steps *REFN* are used before reaching an implementation (*AIMP* here).

ProB [17] is a tool that can execute high-level models described in first-order predicate logic with set theory by translating them into different frameworks, in particular, CLP(FD) and SAT. We use the OscaR library [22] as an additional target bypassing ProB to experiment with different translations into a CP framework. The objective is to extend ProB eventually also with a CP translation. As a result of this approach ProB will be capable of providing an efficient target for the translation of a given model. Furthermore, we obtain a methodology to compare the performance of different frameworks as we can prove that they encode the same model while still exploiting their respective strengths. Finally, beside performance we are also interested in refutation completeness, in particular. Some search heuristics sacrifice refutation completeness in favour of performance. In some practical situations this is acceptable when the alternative is for the search to time out. Of course, for the user of a specification method it is essential to understand the significance of the answer obtained. In this paper we deal with the two latter points. We briefly discuss the translations provided by ProB and the one we use for OscaR. We compare the performance of the different translated models. Furthermore, since ProB can translate models at different levels of abstraction, we can compare performance gains achieved by refining models to lower levels of abstraction.

The following example illustrates the abstraction levels we have to bridge from a high-level problem description to an efficient CP encoding. We use the well-known n-queens problem [3] that is commonly used as a benchmark for constraint solvers and allows evaluating the scalability of our approach. Details are discussed in Section 4.

Example. Consider the following specification of the n-queens problem: n queens need to be placed on a n-times-n board such that no queen can attack another. Let n be a (non-negative) integer constant. Let *NQABS* be the following

predicate, a specification of the n-queens problem in first-order predicate logic:

$$\overbrace{b \subseteq 1\,..\,n \times 1\,..\,n}^{\textit{shape of the board}} \wedge \overbrace{\operatorname{card} b = n}^{\textit{n queens}} \wedge$$
$$\forall p, q \cdot p{\mapsto}q \in b \Rightarrow$$
$$\quad \forall i \cdot i \neq 0 \Rightarrow \underbrace{p{+}i{\mapsto}q \notin b}_{\textit{horizontal}} \wedge \underbrace{p{\mapsto}q{+}i \notin b}_{\textit{vertical}} \wedge \underbrace{p{+}i{\mapsto}q{+}i \notin b}_{\text{“}\backslash\text{”-diagonal}} \wedge \underbrace{p{+}i{\mapsto}q{-}i \notin b}_{\text{“}/\text{”-diagonal}}$$

It captures the main characteristics of the problem: the shape of the board, the number of queens and a constraint specifying that no queen must be placed on the board such that it can be attacked by some queen $p{\mapsto}q \in b$ on the board horizontally, vertically or diagonally.[3]

The following specification *NQCON* is considered an abstract description when the problem is to be solved by means of constraint programming:

$$b \in \text{array } 1\,..\,n \text{ to } 1\,..\,n \wedge \text{allDifferent}\,(b) \wedge$$
$$\text{allDifferent}\,(\lambda x \cdot x \in 1\,..\,n \mid b(x) + x) \wedge \text{allDifferent}\,(\lambda x \cdot x \in 1\,..\,n \mid b(x) - x)$$

where $b \in \text{array } A \text{ to } B$ models an array and is defined to be a total function from A to B (formally, $\operatorname{dom} b = A \wedge \operatorname{ran} b \subseteq B \wedge b^{-1} \,;\, b \subseteq \text{id}$) and allDifferent (b) for an array b as $b \,;\, b^{-1} \subseteq \text{id}$, that is, b is injective. CP frameworks usually have optimised search heuristics for well-known constraints. See the study of efficient solvers for the `allDifferent` constraint in [13], for instance. After all, it requires some reasoning to see that any solution of the n-queens problem described by *NQCON* is a solution to the specification *NQABS*. Refinement bridges the gap between the two abstraction levels. It is tempting to ask for a stronger relationship than refinement by requiring that no solutions are lost in the implementation. However, this can be considered a design decision as one could, e.g., wish to remove certain “unwanted” solutions.

If the approach described in this article is followed, then specification and refinement give rise to a method of multi-paradigm programming where imperative, functional and logic program development styles are mixed. Program refinement integrates imperative and logic programming, whereas the theories used in the proofs contribute functional programming. All proofs presented in this article have been carried out with the Isabelle proof assistant [20] in Isabelle/HOL, that supports this view directly.

Related Work. In [10] the authors argue that specifications should be non-executable, being more expressive as well as providing a higher level of abstraction. Moreover, it is stated that executable specifications, being close to a programming language style, may introduce implementation bias together with over-specification. In general the distinction between these two specification styles is that a non-executable specification describes *what* to achieve, while the other captures *how* to achieve it. Hence, the two styles require a trade-off involving expressiveness against executability. A counterargument to [10] is discussed in [6] showing how abstract specifications can be written to be executable often even

[3] The term $p{\mapsto}q$ denotes the pair with first component p and second component q.

correspond closely to logic programs. The current literature addressing this gap focuses on translations from a formal model towards a constraint solver. For example [5] and [15] presents translations from Z to Prolog and VDM to ABC, respectively. However, such approaches compared to the methodology presented, focuses solely on making specifications executable, and do not consider combining refinement together with limiting the search space. In all of this work executable and non-executable specifications are opposed to each other. In our work specifications that are not readily executable are refined to executable ones. These may be non-deterministic or deterministic.

Overview of this Article. The remainder of this paper is organised as follows. Section 2 describes our approach of specification, refinement and constraint solving. It provides an example of an (admittedly simple) data refinement. This work focuses on methodological issues of combining different formal methods techniques and data refinement is one of the aspects we discuss. Section 3 discusses the translation of specifications into constraint programming languages. It is important to be aware of this because these languages directly affect the specifications that can be executed (just as is the case for other programming languages). In Section 4 we discuss the n-queens problem in greater depth. The n-queens problem is representative of combinatorial problems, in general. Whereas the example from Section 2 makes use of data refinement, the n-queens problem changes the structure of the specification during the refinement. Verified standard constraint programming models are derived by refinement. All the models are executed in ProB to witness the performance gains (or lack thereof). The final implementation is also executed in OscaR. The various models and proofs are available at `https://github.com/miranha/SpecCP`.

2 Refinement and Constraint Programming

Software specifications are described conveniently using predicate logic together with set theory, abstracting from details of data representation and program execution. They are abstract models of software. The discussion of refinement in this section follows conceptually the refinement notion of Event-B [1]. However, we avoid the discussion of Event-B proof obligations and focus on the joint use of refinement and constraint solving. We use first-order logic specifications with set theory and integer arithmetic. Isabelle/HOL, which is used to prove the refinement steps, is also used in other contexts to verify properties of Event-B and VDM specifications [4,25]. The examples that we use in this article, however, we have translated from and to Isabelle by hand into the dialect of the B-notation used by ProB, which can be achieved straightforwardly.

Refinement. In some situations abstract models *SPEC* can be implemented with reasonable effort as deterministic programs *PROG*. Specification *SPEC* is not as efficiently executable as the deterministic implementation *PROG*. Refinement permits us to improve *SPEC* gradually, introducing more implementation details step by step until reaching *PROG*. There are various notions of refinement. Implication is one of them [11]: *REFN* refines *SPEC* if and only

if $REFN \Rightarrow SPEC$. This notion can be extended to permit changes in the data representation, say, replacing variable v by variable w. The relationship is expressed by a predicate $SIMR(w, v)$. This is referred to as data refinement [24]. If $SIMR(w, v)$ is functional of the shape $v = SIMF(w)$, data refinement of $SPEC(v, v')$ by $REFN(w, w')$ can be described as follows:

$$w = SIMG(v) \wedge REFN(w, w') \wedge v' = SIMF(w') \Rightarrow SPEC(v, v')$$

where $w = SIMG(v) \Rightarrow v = SIMF(w)$. Now, $SIMR$ is itself a specification and can be implemented (or it may be an implementation already). If $REFN$, $SIMF$ and $SIMG$ are executable, then we can compute $SPEC$ in terms of them. In order to be able to understand the solution produced by the implementation we can use $SIMF$ and $SIMG$ to translate between the data representations. The refinement relation is considered part of the implementation and implemented itself. So, refinement is simply implication. For instance, we could sort an array a in terms of another representation b,

$$b = SIMG(a) \wedge REFN(b, b') \wedge a' = SIMF(b') \Rightarrow SPEC(a, a') \ .$$

More generally, we refine a specification $SPEC$ by a sequential program $TOCON;REFN;TOABS$, where $TOCON$ translates into a suitable data representation and $TOABS$ translates back. Figure 2 shows how $SPEC$ is refined. In the examples treated in this article we only encounter functions of the sort

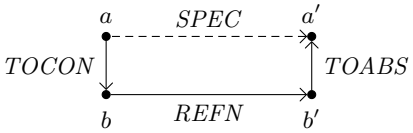

Fig. 2. Specification $SPEC$ refined by specification $TOCON;REFN;TOABS$

$SIMF$ and $SIMG$ as described above.

This notion of refinement corresponds to guard refinement in Event-B where the concrete guard of an event (resp. a state transition) must imply the abstract guard [1]. It also corresponds to postcondition strengthening in VDM [14] or the refinement calculus [2,19]. In all of these cases a predicate is used to specify a successor state. In this article we focus on this predicate and use implication to express refinement. It can be applied easily in various model-based formal methods.

In some cases providing a deterministic implementation may be very difficult and constraint programming can be used. Instead of using refinement to provide algorithmic structure, it can be used to cast a predicate into a shape that can be solved efficiently by a constraint solver.

Example. We illustrate this by means of the puzzle *Who killed Agatha?* [23].

"Someone in Dreadsbury Mansion killed Aunt Agatha. Agatha, the butler, and Charles live in Dreadsbury Mansion, and are the only ones to live there. A killer always hates, and is no richer than his victim. Charles hates no one that Agatha hates. Agatha hates everybody except the butler. The butler hates

everyone not richer than Aunt Agatha. The butler hates everyone whom Agatha hates. No one hates everyone. Who killed Agatha?"

Assume we have three distinct constants *Agatha*, *butler* and *Charles*. We wish to determine the killer k in the following specification $WKA(r, h, k)$

$$\begin{array}{l} Agatha \in (h \setminus r)[\{k\}] \land irreflexive(r) \land transitive(r) \land antisymmetric(r) \land \\ h[\{Agatha\}] \cap h[\{Charles\}] = \varnothing \land h[\{Agatha\}] = P \setminus \{butler\} \land \\ (\forall x \cdot x \mapsto Agatha \notin r \Rightarrow butler \mapsto x \in h) \land h[\{Agatha\}] \subseteq h[\{butler\}] \land \\ (\forall x \cdot h[\{x\}] \neq P) \end{array}$$

where $P = \{Agatha, butler, Charles\}$, r models the relationship *richer* and h models the relationship *hates*. This specification captures the informal description above. The use of symbolic constants improves readability. However, searching for a solution for k this may not necessarily be the most efficient representation. Switching to integer constants $I = 0 \mathinner{..} 2$ instead, we could use them in arithmetic expressions as in $\sum_{y \in h[\{x\}]} y$. We can map the values in I to values in P by means of the function

$$abs(i) = \text{if } i = 0 \text{ then } Agatha \text{ else } (\text{if } i = 1 \text{ then } butler \text{ else } Charles)$$

and from P to I by means of the function

$$con(p) = \text{if } p = Agatha \text{ then } 0 \text{ else } (\text{if } p = butler \text{ then } 1 \text{ else } 2) \ .$$

Note that, $i = con(p) \Rightarrow p = abs(i)$. We can prove that $WKB(s, j, \ell)$ given by

$$\begin{array}{l} 0 \in (j \setminus s)[\{\ell\}] \land irreflexive(s) \land transitive(s) \land antisymmetric(s) \land \\ j \subseteq I \times I \land s \subseteq I \times I \land j[\{0\}] \cap j[\{2\}] = \varnothing \land j[\{0\}] = I \setminus \{1\} \land \\ (\forall x \cdot x \in I \land x \mapsto 0 \notin s \Rightarrow 1 \mapsto x \in j) \land j[\{0\}] \subseteq j[\{1\}] \land (\forall x \cdot x \in I \Rightarrow j[\{x\}] \neq I) \end{array}$$

refines $WKA(r, h, k)$. Formally,

$$\begin{array}{l} s = \{con(x) \mapsto con(y) \mid x \mapsto y \in r\} \land j = \{con(x) \mapsto con(y) \mid x \mapsto y \in h\} \land \\ WKB(s, j, \ell) \land k = abs(\ell) \Rightarrow WKA(r, h, k) \ . \end{array}$$

Note, that the equations in the first line describe functions. Finally, we arrive at a shape of the specification $WKC(s, j, \ell)$ that permits efficient execution by a constraint solver,

$$\begin{array}{l} \ell \in I \land 0 \mapsto \ell \in j^{-1} \land 0 \mapsto \ell \notin s^{-1} \land \\ irreflexive(s) \land transitive(s) \land antisymmetric(s) \land \\ j \subseteq I \times I \land s \subseteq I \times I \land (\forall x \cdot x \in I \land 0 \mapsto x \in j \Rightarrow 2 \mapsto x \notin j) \land \\ 0 \mapsto 0 \in j \land 0 \mapsto 1 \notin j \land 0 \mapsto 2 \in j \land (\forall x \cdot x \in I \land x \mapsto 0 \notin s \Rightarrow 1 \mapsto x \in j) \land \\ (\forall x \cdot x \in I \land 0 \mapsto x \in j \Rightarrow 1 \mapsto x \in j) \land (\forall x \cdot x \in I \Rightarrow \sum_{y \in j[\{x\}]} 1 \leq 2)) \end{array}$$

To translate this formula into a dedicated constraint programming model, we would next replace some formulas to match with the library of the constraint programming language. For example, we would replace $\sum_{y \in j[\{x\}]} 1$ by $\text{sum}(j[\{x\}])$

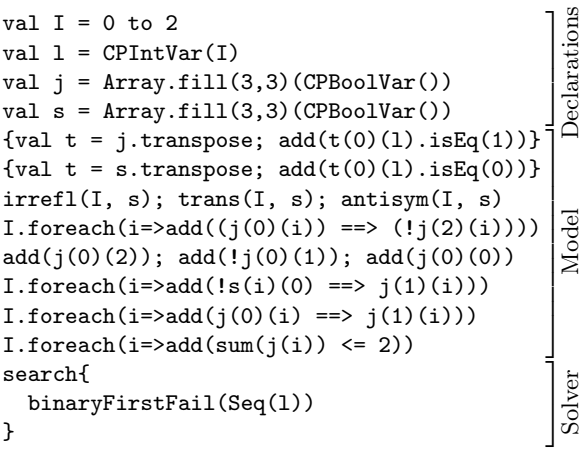

Fig. 3. OscaR model with solver for the puzzle "Who killed Agatha?"

in the last row. Furthermore, we would replace the relations j and s by corresponding two-dimensional boolean arrays. However, this representation is a trivial rewriting of the formula above, replacing terms of the shape $(x, y) \in z$ by $a(x, y) = \text{TRUE}$, and we do not show it. Figure 3 shows how the model for the puzzle could be translated to OscaR. The translation has been produced by hand following the rules described in Section 3. It consists of a section of declarations, the actual model and a call to the solver. The most interesting part for this article is the development of models that can be analysed efficiently. Note, the use of `transpose` in the OscaR model. This is an implementation concern that is reflected by the use of the relational inverse in the abstract formula: the first index of `j` and `s` must not be a `CPIntVar`.

The constraint solver of ProB can solve the models at all abstraction levels that we have presented in this section. A set like $P = \{Agatha, butler, Charles\}$ gets translated by ProB internally into the set $\{1, 2, 3\}$; as such the first data refinement *WKB* is performed internally by ProB and not required by the user. The translation of the constraint $\forall x \cdot h[\{x\}] \neq P$ into a sum constraint in *WKC* is also not necessary: ProB has built-in support for set equality and inequality and can handle the set inequality $h[\{x\}] \neq P$ relatively efficiently.

3 Translation and Constraint Solving

This section discusses the translation approaches introduced in Section 1. Section 3.1 discusses the use of specifications as constraint solving languages directly with ProB, outlining some important concepts of the applied translations. Section 3.2 outlines the translation of predicate logic statements from a specification into a language for constraint solving in OscaR. Whereas the translation to OscaR requires the user to adopt certain data representations, ProB can also solve the abstract specifications. In fact, internally ProB attempts to find efficient representations to increase the speed of the search. However, no generally efficient method exists [12]. A longer term goal of our work combining refinement with

translations is to permit writing specifications that steer the internal representations to gain more control over the efficiency while keeping the abstraction level high. Independently of this, the approach used by ProB for finding the solution to a non-deterministic specification influences the specification style for specific problems. In other words, implementation concerns always shine through.

3.1 Translation in ProB

A key concept of ProB's default constraint solver [8] is reification, i.e., representing the truth value of a constraint C by a boolean decision variable $R_C \in 0..1$ so that $R_C = 1 \Leftrightarrow C$. Reification is important for ProB to avoid choice points, e.g., for $P \vee Q$, ProB will set up the constraint similar to $R_P \in 0..1 \wedge (R_P = 1 \Leftrightarrow P) \wedge R_Q \in 0..1 \wedge (R_Q = 1 \Leftrightarrow Q) \wedge (R_P = 1 \vee R_Q = 1)$. The constraint $(R_P = 1 \vee R_Q = 1)$ is handled by ProB's boolean constraint solver, while P and Q can be handled by different solvers. Indeed, arithmetic predicates and operators like $x + y \geq 0$ are mapped to the finite domain solver CLP(FD). Equality, inequality, set membership and subset constraints are handled by a dedicated solver in ProB itself.

In the worst case, universal quantifiers $\forall x.Q \Rightarrow R$ get expanded when the domain $\{x|Q\}$ is known. However, there is special support for $\forall x.x \in S \Rightarrow P(x)$: here P will be checked for every element added to S, even when S is not yet fully known. Certain universal quantifiers can be expanded into a conjunction: e.g., $\forall x.x \in 1..3 \Rightarrow P(x)$ gets automatically translated into $P(1) \wedge P(2) \wedge P(3)$; this enables reification of the entire quantified predicate. The treatment of existential quantifiers is similar; in the worst case they are evaluated when all but the quantified variables are known. However, some quantifiers can be expanded into disjunctions: e.g., $\exists x.x \in 1..3 \wedge P(x)$ gets translated into $P(1) \vee P(2) \vee P(3)$.

As mentioned earlier, ProB provides alternate constraint solving backends: a translation to SAT via Kodkod and a translation to SMTLib using Z3. The SAT encoding via Kodkod performs well for constraints over relations and operators such as relational composition and transitive closure. It, however, requires all base types to be finite. For the "Who killed Agatha?" of Section 2 the SAT backend is slightly slower (60 ms vs 10 ms). For the n-queens problem in Section 4, it is considerably slower than ProB's default solver (e.g., over 5 seconds compared to 10 ms for $n = 16$). The Z3 backend is even slower (e.g., over 6 seconds for $n = 8$). Hence, in the rest of the paper we have concentrated on the default solver of ProB.

3.2 Translation to OscaR

OscaR (using many ideas of Comet [12]) supports the execution of non-deterministic specifications, as well as permitting us to experiment easily with different model representations and search heuristics. Furthermore, OscaR is a library for Scala [21], so the used syntax is Scala syntax with the extensions made by OscaR. A specification $\forall z \cdot p_1(x, z) \wedge \ldots p_n(x, z)$ specifies values of variables x to be computed, constrained by relationships among each other and variables z.

Universal quantifiers and conjunctions are preferred over existential quantifiers and disjunctions as the latter may lead to backtracking. However, some uses do lead to loss of efficiency and sometime even lead to performance improvements. For instance, we use auxiliary variables to add additional constraints (sometimes redundant but with an effect on performance). So a specification has the form $\exists y \cdot \forall z \cdot p_1(x, y, z) \wedge \ldots p_n(x, y, z)$ where y corresponds to an auxiliary variable of the corresponding constraint program. Generally, OscaR tracks domains of variables efficiently, while the search algorithms restrict the domains further in order to minimise the remaining non-deterministic choices. By means of backtracking OscaR is able to find multiple solutions. In comparison to OscaR, ProB is able to find solutions for certain infinite problems and it will find all solutions for finitely posed problems. For OscaR the latter property depends on the chosen search heuristics. However, usually completeness is sacrificed for efficiency.

The OscaR CP solver adopts a modelling methodology using decision variables and constraints among them, similar to other CP solvers. Its general structure is based around three components as shown in Figure 3: a declaration of the decision variables, a model and a search heuristic. The *declaration* introduces the decision variables and their domains. The *model* component captures the constraints for the decision variables. The *search heuristic* specifies a non-deterministic search heuristic for finding a solution for the model. Usually the declaration and model together are considered to be one component. We treat them separate as this is relevant for the translation. The domains associated with the different variables are highly relevant for the efficiency of the search heuristic. We only discuss the translation for the declaration and the model, and assume that a common search heuristic is to be applied to the model, such as *binary first fail*. The translation is indicated by the notation $S \overset{cp}{\rightsquigarrow} P$ describing the translation from specification element S to constraint programming construct P. We describe the translation by way of examples. They are easy to understand and generalise. For these translations the focus is on the extensions relevant for a specification as provided by OscaR.

The Declarations. The basic data types allowed for decision variables supported by OscaR are boolean and integer types. An integer type `CPIntVar` must be given a finite domain from which its values may be drawn. The boolean type `CPBoolVar` is a subtype of integer with the domain of 0 and 1. The translation of integer and boolean domains as well as declaration of decision variables can be translated as follows, respectively: $D = m \mathinner{..} n \overset{cp}{\rightsquigarrow}$ `val D = m to n`, $i \in D \overset{cp}{\rightsquigarrow}$ `val i = CPIntVar(D)` and $b \in \mathbb{B} \overset{cp}{\rightsquigarrow}$ `val b = CPBoolVar()`.

The Model. Once the decision variables have been declared together with their domain, constraints can be added to the constraint store. In OscaR constraints are added to the constraint store by means of the function `add(...)`. When adding constraints, two important aspects of the constraints to consider are conjunctions and disjunctions. Although both are of boolean type, they are treated differently by OscaR to improve the performance of the search for a solution. Atomic predicates are translated by $P \overset{cp}{\rightsquigarrow}$ `add(`$\mathtt{P}^{cp}$`)` where $\mathtt{P}^{cp}$ is the atomic predicate in OscaR syntax, for instance, the translation of $x \leq y$ is `x.isLeEq(y)`.

A conjunction describes a collection of constraints each of which needs to be true. They can be added separately to the constraint store. For example, $x = 5 \wedge y \leq x \overset{cp}{\rightsquigarrow}$ `add(x.isEq(5)); add(y.isLeEq(x))`. This is the common approach for treating conjunctions in OscaR. The general form of this is of the following shape $P \wedge Q \overset{cp}{\rightsquigarrow} P'$ `;` Q' where P and Q are translated to P' and Q'.

A disjunction describes a collection of constraints either of which needs to be true. It cannot be divided into separate constraints as is the case with conjunction. All separate constraints are implicitly conjoined. For this reason disjunctions are represented as boolean expressions within the constraint language. The operators available for this are logical or (`||`), logical and (`&&`) as well as implication (`==>`). This lifts the predicate into a constraint expression of the OscaR constraint language. This is applied to the disjunction translation, for example, $x = 5 \vee y \leq x \overset{cp}{\rightsquigarrow}$ `add(x.isEq(5) || y.isLeEq(x))`, while the general form is $P \vee Q \overset{cp}{\rightsquigarrow}$ `add(`$\mathtt{P}^{cp}$ `||` $\mathtt{Q}^{cp}$`)` where $\mathtt{P}^{cp}$ and $\mathtt{Q}^{cp}$ are translations of the conjuncts into OscaR syntax. Technically, $\mathtt{P}^{cp}$ and $\mathtt{Q}^{cp}$ require a second layer in the translation for constraint expressions but we do not develop this here. We make the strong assumption that the disjuncts are atomic. This introduces an implementation concern into the constraint modelling language. It is justified by the following reasoning: representing operators within constraints creates an additional layer for the search reducing its performance.[4] So the larger portion of a specification contains such constructs, the lower the performance will be. Hence, generally it is discouraged to represent large parts of a constraint program as constraint expressions because this negatively impact implementation performance. Nonetheless, the connectives `||`, `&&` and `==>` are very useful in OscaR (see Fig. 3) but should be used as little as possible. In particular, for conjunction it can usually be avoided.

We could add the rule $P \wedge Q \overset{cp}{\rightsquigarrow}$ `add(`$\mathtt{P}^{cp}$ `&&` $\mathtt{Q}^{cp}$`)` to the translation, e.g. to deal with cases where conjunctions appear within disjunctions. However, we are more interested to use refinement to arrive at efficient representations than at maximising the class of translatable specifications. In any case, ProB which is described above permits a large class of translations already. The objective of the translation to OscaR is to achieve high performance. Generality is secondary. Whenever possible, disjunctions should be avoided. For instance, instead of writing a specification $x = 1 \vee x = 2 \vee x = 3$ one can declare the domain of x correspondingly. If that is not possible, one could also introduce a decision variable y with domain $1..3$ and use the constraint $x = y$. Computations concerning constraints are very efficient! Of course, there will be cases where disjunctions are unavoidable. Negation is available because the domains of the involved decision variables are declared. As a result, negation may never lead to infinite domains rendering the model non-executable. The specification must contain a term $x \in D$ providing the (finite) domain for each decision variable x. And each universal quantification must provide a (finite) domain, that is, it must be of the shape $\forall x \cdot x \in D \Rightarrow P$. Universal quantification is discussed next.

[4] Private communication with Pierre Schaus.

A universal quantifier can be translated by adding individually each constraint described by way of the quantified variable with finite domain. For example, $\forall x \cdot x \in 4 \mathinner{..} 7 \Rightarrow x > 3 \overset{cp}{\rightsquigarrow}$ `(4 to 7).foreach(x => add(x>3))`. The Scala operator `foreach` iterates through all elements of the range (here `(4 to 7)`). The general form of the translation has the shape $\forall x \cdot x \in m \mathinner{..} n \Rightarrow P(x) \overset{cp}{\rightsquigarrow}$ `(m to n).foreach(x =>` $P'(x)$`)`, where $P'(x)$ is the translation of $P(x)$. Universal quantification is essentially treated like an indexed conjunction.

Finally, existential quantifiers are discussed next. If existential terms are involved in a specification they must be lifted so that the specification has the shape $\exists y \cdot p_1(x, y) \wedge \ldots p_n(x, y)$ so that they can be treated as auxiliary variables. The translation does not support existential quantifiers occurring inside universal quantifiers. OscaR does not support this directly. In principle, the translation could deal with this in the way ProB does (see the discussion in Section 3.1). Whereas in the case of the universal quantifier the bound variable gets eliminated in the translation, in the case of existential quantifiers they stay. They are treated like the global variables but are not considered part of the result. As indicated in the introduction of this section, they should be considered auxiliary variables.

4 Solving the N-Queens Problem

In the example "Who killed Agatha?" of Section 2 we demonstrated the change of data representation. This permitted us to express a subset relationship as an arithmetic expression. The problem statement remained structurally unchanged: it is easy to see how the computed solution is related to the abstract specification.

In this section we change the specification structurally while keeping the data representation by way of the n-queens problem. The encodings of the n-queens problem are not new. Usually, one finds informal arguments that argue why a certain encoding is correct (e.g. [12]). However, the smarter the encodings get, the more likely errors occur in the corresponding models. This is no different from the situation in sequential programming. Finally, we evaluate the various models and compared there scalability.

Refinement of n-queens. The initial specification describes the problem in terms of the geometry of the chess board. Numbering the rows and columns from 1 to n for some n larger than 0, we can describe the queens on which fields a queen could attack arithmetically as indicated in Fig. 4. We can express that a

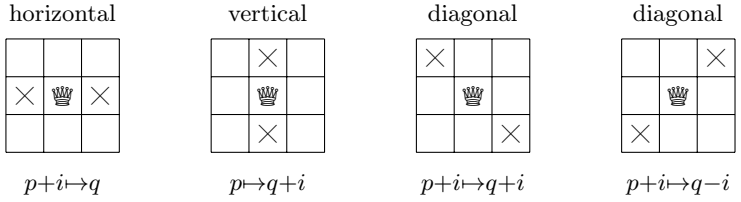

Fig. 4. Positions "×" on 3×3-board that can be attacked by a queen "♕" on position $p \mapsto q$ (where $i \in \{-1, 1\}$)

queen at position $p \mapsto q$ cannot attack another queen by requiring that no queen

may be placed on a position it may reach, that is, $p{\mapsto}q \in b$ implies

$$\forall i \cdot i \neq 0 \Rightarrow p{+}i{\mapsto}q \notin b \wedge p{\mapsto}q{+}i \notin b \wedge p{+}i{\mapsto}q{+}i \notin b \wedge p{+}i{\mapsto}q{-}i \notin b \ ,$$

that no other queen is on a position that can be attacked by it. The complete specification *NQABS* also specifying the board positions $1 \mathinner{..} n \times 1 \mathinner{..} n$ and the number of queens n to be placed on the board is easy to understand,

$$\begin{array}{l} b \subseteq 1 \mathinner{..} n \times 1 \mathinner{..} n \wedge \mathrm{card}\, b = n \wedge \forall p, q \cdot p{\mapsto}q \in b \Rightarrow \\ \quad \forall i \cdot i \neq 0 \Rightarrow p{+}i{\mapsto}q \notin b \wedge p{\mapsto}q{+}i \notin b \wedge p{+}i{\mapsto}q{+}i \notin b \wedge p{+}i{\mapsto}q{-}i \notin b \end{array}$$

and relate to the informal statement of the n-queens problem. It relates to the board geometry, the number of queens and the positions a queen placed on the board may attack. Unfortunately, this specification is not efficient to execute. The first problem we identify is the size of the board to consider. It is n^2. Because a queen can attack any other queen that is placed in the same column we can exclude all configurations of queens on a board where more than one queen is on any column. Thus, there must be precisely one queen on each column. Hence, we can represent the board as an array in a first refinement *NQARR*

$$\begin{array}{l} b \in \mathrm{array}\ 1 \mathinner{..} n \ \mathrm{to}\ 1 \mathinner{..} n \wedge \\ \forall p, q \cdot p{\mapsto}q \in b \Rightarrow \forall i \cdot i \neq 0 \Rightarrow p{+}i{\mapsto}q \notin b \wedge p{+}i{\mapsto}q{+}i \notin b \wedge p{+}i{\mapsto}q{-}i \notin b \end{array}$$

Of course, it is no longer necessary to verify that at most one queen placed in each column. The predicate $\forall p, q \cdot p{\mapsto}q \in b \Rightarrow \forall i \cdot i \neq 0 \Rightarrow p{+}i{\mapsto}q \notin b$ is a slightly complicated way of saying that b is injective. This is expressed by the formula allDifferent (b). When translating to constraint programming languages like OscaR, predicates like allDifferent are translated into efficient representations in the constraint store. Typically, a library of such predefined constraint predicates exists in constraint programming languages that permit performance improvements when used. The only predicate that could still be improved is $\forall p, i \cdot i \neq 0 \Rightarrow p{+}i{\mapsto}b(p){+}i \notin b \wedge p{+}i{\mapsto}b(p){-}i \notin b$ where we have used that b is a total function from $1 \mathinner{..} n$ to $1 \mathinner{..} n$ to rewrite the predicate. We have

$$\begin{array}{ll} & \forall p, i, q \cdot i \neq 0 \Rightarrow p{+}i{\mapsto}b(p){+}i \neq q{\mapsto}b(q) \wedge p{+}i{\mapsto}b(p){-}i \neq q{\mapsto}b(q) \\ \Leftarrow & \forall p, i, q \cdot i \neq 0 \Rightarrow i{\mapsto}i \neq q{-}p{\mapsto}b(q){-}b(p) \wedge i{\mapsto}{-}i \neq q{-}p{\mapsto}b(q){-}b(p) \\ \Leftarrow & \forall p, i, q \cdot i \neq 0 \Rightarrow i{\mapsto}i \neq \mathrm{abs}\,(q{-}p){\mapsto}\mathrm{abs}\,(b(q){-}b(p)) \\ \Leftarrow & \forall p, q \cdot p \neq q \Rightarrow \mathrm{abs}\,(q{-}p) \neq \mathrm{abs}\,(b(q){-}b(p)) \qquad (1) \end{array}$$

Hence, *NQARR* is refined by *NQMID* which we define by

$$\begin{array}{l} b \in \mathrm{array}\ 1 \mathinner{..} n \ \mathrm{to}\ 1 \mathinner{..} n \wedge \\ \mathrm{allDifferent}\,(\mathrm{b}) \wedge \forall p, q \cdot p \neq q \Rightarrow \mathrm{abs}\,(q{-}p) \neq \mathrm{abs}\,(b(q){-}b(p)) \end{array}$$

Specification *NQMID* is often used in examples for constraint programming. Furthermore, we have $(\forall p, q \cdot p \neq q \Rightarrow b(p) + p \neq b(q) + q \wedge b(p) - p \neq b(q) - q)$ $\Rightarrow$ (1). Thus we can refine *NQMID* by *NQCON*, defined by

$$\begin{array}{l} b \in \mathrm{array}\ 1 \mathinner{..} n \ \mathrm{to}\ 1 \mathinner{..} n \wedge \mathrm{allDifferent}\,(b) \wedge \\ \mathrm{allDifferent}\,(\lambda x \cdot x \in 1 \mathinner{..} n \mid b(x) + x) \wedge \mathrm{allDifferent}\,(\lambda x \cdot x \in 1 \mathinner{..} n \mid b(x) - x) \end{array}$$

The predicate allDifferent (a) is widely used in constraint programming. Informally, it is defined as: no value occurs more than once in the array a.

A formal refinement verifies that any solution computed for a refined specification is also a solution of the initial specification. In addition, the proofs provide an explanation for the correctness of the encodings.

Note that the way we have proved the refinements, we have not assured the existence of a solution in the refined specifications. We could refine any specification by *false*. One could prove the existence of a solution for each specification. However, for specific values of n this is what a constraint solvers does: it finds values for the decision variables that satisfy the specification.

Performance of the Specifications. We evaluate the performance of all four specifications *NQABS*, *NQARR*, *NQMID* and *NQCON*, where ProB can solve all while OscaR is only applied to *NQCON* using its built-in `allDifferent` constraint. Additionally for ProB, we add *NQPRM* which uses a built-in predicate for permutations and a random search heuristic and is otherwise identical to *NQCON*. The specifications are evaluated for various board sizes ranging from $n = 8$ to $n = 1000$, with a step size of 8. Benchmarks were run on AMD Opterons with 2 GHz and four physical cores; up to three benchmarks were run in parallel. Measurements show the time in seconds taken to find one solution for the corresponding specification of the board size n. Results are shown in Figure 5, where missing combinations of boards size and solver are due to time-outs, i.e., the solver failed to produce an answer in 5 minutes.

All in all, benchmark results support our motivation to translate specifications to OscaR, as OscaR allows to explorate much larger models. First, note that the most abstract specification *NQABS* only scales to $n = 8$. Next, the more low-level *NQARR* and its following refinement *NQMID* display similar performance and scale to at most $n = 168$. The least abstract model *NQCON* scales to $n = 600$, while the random permutation version *NQPRM* only scales slightly better. Finally, the OscaR version solves a board size of up to $n = 1000$.

Additionally, note that OscaR displays a somewhat erratic behaviour, e.g., it is slower or cannot find a solution for certain smaller board sizes, while being able to solve larger ones more efficiently. This might be due to both the internal implementation of OscaR and the `allDifferent` constraint, allowing to exploit symmetries for certain board sizes. For example the search of the state space might be vulnerable to wrong choices, such that it might has to explore a large sub-tree before getting on the right track again. With techniques such as conflict driven clause learning [27] and random restarts [7], the search can become much more resilient. In this paper OscaR and ProB are used as representatives of a CP solver, and the presented method can consider others as well. For this purpose, the n-queens problem served well as a simple, yet scalable, benchmark for evaluating the methodology presented. As we have argued in [16], more involved benchmarks allow for higher transferability of the benchmarks to real-world applications. In this respect, in [9,26] we discuss real-world applications, where constraint solving is used for solving and validating larger timetables and railway network configurations.

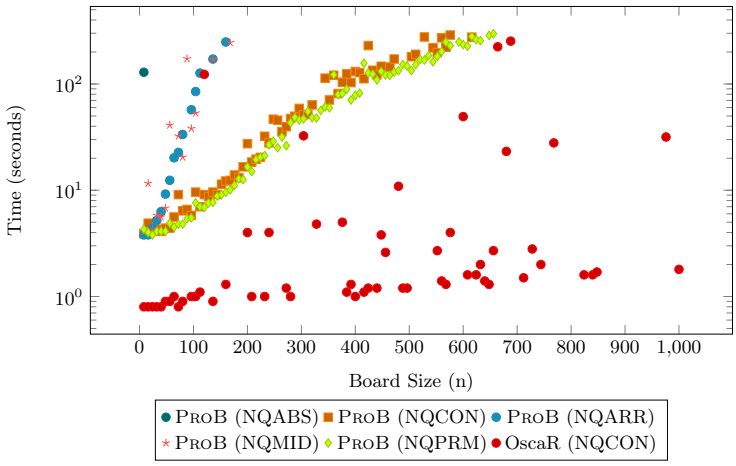

Fig. 5. Results of successful benchmarks, time-outs omitted

5 Conclusion

We have shown how refinement permits work with executable specifications without committing early to implementation-biased data representations. Formal specification should be considered abstract programming and associated reasoning techniques like refinement support for implemention. In principle, this software is multi-paradigm: it may contain imperative, logic and functional parts that appear seamless in abstract specifications. Compared to work like [19] where *programs* are executable or non-executable, we change the focus once more and say that *specifications* are executable or non-executable to emphasise the abstraction also in the final implementation. We have described a translation to OscaR that we use for experimenting with translations and search heuristics. This complements the use of ProB where making consistent changes to the translation or search is more intricate. The preliminary evaluation to compare efficiency of abstraction levels points towards the direction in which this research will be continued: specifications are the better programs!

Acknowledgments. The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047.

References

1. Abrial, J.R.: Modeling in Event-B – System and Software Engineering. Cambridge University Press (2010)
2. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Springer (1998)
3. Bruen, A., Dixon, R.: The n-queens problem. Discrete Math. 12(4), 393–395 (1975)

4. Couto, L.D., Foster, S., Payne, R.J.: Towards verification of constituent systems through automated proof. CoRR abs/1404.7792 (2014)
5. Dick, A.J.J., Krause, P.J., Cozens, J.: Computer Aided Transformation of Z into Prolog, pp. 71–85. Springer (1990)
6. Fuchs, N.E.: Specifications are (preferably) executable. Software Engineering Journal 7(5), 323–334 (1992)
7. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting Combinatorial Search Through Randomization. In: Mostow, J., Rich, C. (eds.) AAAI. pp. 431–437. AAAI Press / MIT Press (1998)
8. Hallerstede, S., Leuschel, M.: Constraint-based deadlock checking of high-level specifications. TPLP 11(4–5), 767–782 (2011)
9. Hansen, D., Schneider, D., Leuschel, M.: Using B and ProB for Data Validation Projects. In: Butler, M.J., Schewe, K.D., Mashkoor, A., Biró, M. (eds.) ABZ. LNCS, vol. 9675, pp. 167–182. Springer (2016)
10. Hayes, I., Jones, C.B.: Specifications are not (necessarily) executable. Softw. Eng. J. 4(6), 330–338 (1989)
11. Hehner, E.C.R.: A Practical Theory of Programming. Springer (1993)
12. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. MIT Press (2009)
13. van Hoeve, W.J.: The alldifferent constraint: A survey. arXiv cs/0105015 (2001)
14. Jones, C.B.: Systematic Software Development using VDM. Prentice Hall (1990)
15. Kans, A., Hayton, C.: Using ABC to Prototype VDM Specifications. SIGPLAN Not. 29(1), 27–36 (1994)
16. Krings, S., Leuschel, M., Körner, P., Hallerstede, S., Hasanagic, M.: Three Is a Crowd: SAT, SMT and CLP on a Chessboard. In: Calimeri, F., Hamlen, K.W., Leone, N. (eds.) PADL. LNCS, vol. 10702, pp. 63–79. Springer (2018)
17. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT 10(2), 185–203 (2008)
18. Leuschel, M., Schneider, D.: Towards B as a high-level constraint modelling language – solving the jobs puzzle challenge. In: Ameur, Y.A., Schewe, K.D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 101–116. Springer (2014)
19. Morgan, C.C.: Programming from specifications. Prentice Hall, 2nd edn. (1994)
20. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag (2002)
21. Odersky, M., al.: An Overview of the Scala Programming Language. Tech. Rep. IC/2004/64, EPFL, Lausanne, Switzerland (2004)
22. OscaR Team: OscaR: Scala in OR (2012), available from `bitbucket.org/oscarlib`
23. Pelletier, F.J.: Seventy-five problems for testing automatic theorem provers. J. Autom. Reasoning 2, 191–216 (1986)
24. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-oriented Proof Theories and their Comparison, vol. 46. Cambridge University Press (1998)
25. Schmalz, M.: Term rewriting in logics of partial functions. In: Qin, S., Qiu, Z. (eds.) ICFEM. LNCS, vol. 6991, pp. 633–650. Springer (2011)
26. Schneider, D., Leuschel, M., Witt, T.: Model-Based Problem Solving for University Timetable Validation and Improvement. In: Bjørner, N., de Boer, F.S. (eds.) FM. LNCS, vol. 9109, pp. 487–495. Springer (2015)
27. Silva, J.P.M., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)