AlleAlle: Bounded Relational Model Finding with Unbounded Data

Jouke Stoel
stoel@cwi.nl
CWI, Amsterdam
The Netherlands

Tijs van der Storm
storm@cwi.nl
CWI, Amsterdam
University of Groningen, Groningen
The Netherlands

Jurgen J. Vinju
vinju@cwi.nl
CWI, Amsterdam
TU/e, Eindhoven
The Netherlands

Abstract

Relational model finding is a successful technique which has been used in a wide range of problems during the last decade. This success is partly due to the fact that many problems contain relational structures which can be explored using relational model finders. Although these model finders allow for the exploration of such structures they often struggle with incorporating the non-relational elements.

In this paper we introduce AlleAlle, a method and language that integrates reasoning on both relational structure and non-relational elements —the data— of a problem. By combining first order logic with Codd's relational algebra, transitive closure, and optimization criteria, we obtain a rich input language for expressing constraints on both relational and scalar values.

We present the semantics of AlleAlle and the translation of AlleAlle specifications to SMT constraints, and use the off-the-shelf SMT solver Z3 to find solutions. We evaluate AlleAlle by comparing its performance with Kodkod, a state-of-the-art relational model finder, and by encoding a solution to the optimal package resolution problem. Initial benchmarking show that although the translation times of AlleAlle can be improved, the resulting SMT constraints can efficiently be solved by the underlying solver.

CCS Concepts • **Theory of computation** → *Constraint and logic programming*; • **Software and its engineering** → *Specification languages.*

Keywords First order logic, relational algebra, SMT solvers, model finding, constraint problems, constraint optimization problems

Onward! '19, October 23–24, 2019, Athens, Greece

ACM ISBN 978-1-4503-6995-4/19/10…$15.00
https://doi.org/10.1145/3359591.3359726

ACM Reference Format:
Jouke Stoel, Tijs van der Storm, and Jurgen J. Vinju. 2019. AlleAlle: Bounded Relational Model Finding with Unbounded Data. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '19), October 23–24, 2019, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3359591.3359726

1 Introduction

In the last decades relational modeling and model finding has been used to solve problems in a wide range of domains, from security [8], program verification and testing [14, 16], to enterprise modeling [3].[1] Since many computational problems have relational structures relational model finding has shown to be a powerful and useful method. But there is also a large class of problems that is not purely relational and requires reasoning over other attributes as well.

Consider for instance, a simple file system. This structure can be naturally expressed as a relational problem. However, adding constraints on properties like the *depth* or the *size* of file system nodes is not straightforward, or cannot be solved efficiently. In this paper we propose AlleAlle, a language that allows users to model both the relational and the non-relational elements —the data— of their problem.

AlleAlle combines first order logic, Codd's relational algebra (projection, restriction, renaming and natural join) [10], and (reflexive) transitive closure in a single formalism. AlleAlle specifications can be translated to SMT formulas which in turn can be solved by an off-the-shelf SMT solver, such as Z3 [22]. We implemented these ideas in a prototype tool.[2]

Next to solving Constraint Satisfaction Problems (CSP), AlleAlle can be used to solve Constraint Optimization Problems (COP) (cf. traveling salesman). This is achieved by extending the syntax of AlleAlle with the ability to express *optimization objectives* on relations. These optimization criteria are added to the translated SMT formulas and can be solved using Z3's built-in optimization solver *νZ* [7].

We perform an initial performance benchmark and evaluate AlleAlle's expressiveness on a well-known problem in software engineering: optimal package resolution [1]. This

[1] For a overview of the different areas where relational model finding has been applied visit http://alloytools.org/citations/case-studies.html.
[2] https://github.com/cwi-swat/allealle

problem, faced by software package managers, can be compactly expressed as a relational problem in AlleAlle and we show that the resulting SMT formula can be efficiently solved by the underlying SMT solver.

The contributions of this paper can be summarized as follows:

- AlleAlle, a language combining Codd's relational algebra with first order logic, transitive closure, and optimization objectives (Section 3).
- A translation semantics expressed by compiling AlleAlle specifications to SMT constraints (Section 4).
- Initial performance benchmarking of AlleAlle, including a realistic benchmark based on the optimal dependency resolution problem (Section 5).

We conclude the paper with a discussion of related work (Section 6), and an outlook towards future work (Section 7).

2 AlleAlle

AlleAlle is an intermediate language similar to Kodkod's [31] internal model. As such it is aimed at being a target language for high-level relational modeling languages such as Alloy [15, 30]. Instead of using SAT solvers to solve relational constraints, however, AlleAlle leverages native data theories built into SMT solvers, such as Z3. Because of this, AlleAlle can support constraints over unbounded data such as integers, reals, and strings, without having to encode such data values into boolean propositions. As a result, relational specifications employing constraints over data do not suffer from exponential blow-up problems that may occur, for instance, when using fixed bit-width integers in Alloy or Kodkod. In other words, the solving power of AlleAlle is a super-set of that of Kodkod.

AlleAlle is designed to be extensible. Our current implementation supports native integer constraints, but the design of the language and the translation to SMT constraints allows support for other theories (e.g., reals, strings, etc.) in the same way as the current prototype supports integer constraints. Below we illustrate how AlleAlle combines Codd's relational algebra and unbounded data constraints using the example of a file system specification.

2.1 Modeling a File System in AlleAlle

Imagine that we would like to model a new kind of file system and we want to test our design before building the new system. Our new simple file system would have the following structural constraints: it may contain both directories and files, it only has one root, there can be no cyclic dependencies and everything must be reachable from the root. The file system does not allow symbolic links (preventing cyclic references).

Next to these structural constraints we also have some non structural constraints namely, every file must have a positive size; the size of a directory derives from the size of its contents. Finally, every file and directory has a depth which encodes the distance from the root in the hierarchy.

To check these constraints we create an AlleAlle specification that encodes the above constraints, as shown in Listing 1. In the next paragraphs we will explain the different parts of this specification.

Declaring relations The first part, lines 1 to 4, contains the declarations of the relations. Every relation declaration has three parts: the name of the relation, its header, and its tuple bounds. In AlleAlle all relations are bounded meaning that all the tuples that are potentially part of the relation are defined in its upper bound.

For instance, the `File` relation on line 1 has three attributes which are defined in its header: `oid`, `depth` and `size`. The attribute `oid` is of the **`id`** domain while `depth` and `size` are of the **`int`** domain. The **`id`** domain is a bounded domain of arbitrary chosen labels, or *atoms*. The domain contains exactly those values as specified in the relation declarations of the specification. For instance, for this specification the **`id`** domain consists of `f0,f1,f2,d0,d1 and d2`.

The right hand side of the relation declaration lists the tuple bounds. These encode the tuples that can be part of a relation. The `File` relation contains both a lower bound (the tuple set after the `>=` sign), and an upper bound (the tuple set after the `<=` sign). Every relation must have an upper bound. Lower bounds are optional.

Lower bounds can be used to encode partial solutions. They encode the tuples that *must* be part of every satisfying instance. In our example we see that the lower bound of the `File` relation has one tuple, ⟨`oid: f0`, `depth: 2`, `size: 100`⟩. This means that in every satisfying instance found by the solver the relation `File` must at least contain this tuple. In other words, we specify that our file system always must have the file (identified by) `f0` with a size of `100` and two steps removed from the root (`depth: 2`).

The upper bound, on the other hand, contains the tuples that *may* be part of a satisfying instance. For the `File` relation this means that two more tuples may be part of any satisfying instance. Both of these tuples contain question marks for the `depth` and `size` attributes. These question marks —or *holes*— in the tuple definition indicate that the value can be freely assigned by the solver as long as the values satisfy the specification. Holes can only be introduced for non-**`id`** attributes. Attributes of the **`id`** domain always need a value assigned.

The lower and upper bounds of the `Root` relation (line 3) are equal to each other and contain a tuple set with only one tuple, `{<d0>}`. When the lower and upper bounds of a relation are equal, the `=` sign is used to define the exact bound. As a result, in every possible satisfying instance this relation must contain exactly these tuples and not more. The `Dir` relation only has an upper bound (line 2). This indicates that,

```
1  File (oid:id, depth:int, size:int) >= {<f0,2,100>} <= {<f0,2,100>,<f1,?,?>,<f2,?,?>}
2  Dir (oid:id, depth:int, size:int)  <= {<d0,?,?>,<d1,?,?>,<d2,?,?>}
3  Root (oid:id)                       = {<d0>}
4  contents (from:id, to:id)          >= {<d0,d1>} <= {<d0,d0>,<d0,d1>..<d2,d2>,<d0,f0>..<d2,f2>}
5
6  // Contents is a relation that goes from Dir -> (Dir+File)
7  contents in (Dir[oid as from][from] x (Dir + File)[oid as to][to])
8  // A dir cannot contain itself
9  forall d : Dir[oid] | no d[oid as to] & (d[oid as from] |x| ^contents)[to]
10 // Root is a Dir
11 Root in Dir[oid]
12 // All files and dirs are (reflexive-transitive) 'content' of the Root dir
13 (File[oid] + Dir[oid])[oid as to] in (Root[oid as from] |x| *contents)[to]
14 // All files and dirs can only be contained by one dir
15 forall f : (File + Dir)[oid] | lone contents |x| f[oid as to]
16
17 // All files have a positive size
18 forall f : File | some f where size > 0
19
20 // The size of a dir is the sum of all files that are transitively part of this directory
21 forall d : Dir |
22   let containedFiles = (d[oid][oid as from] |x| ^contents)[to][to as oid] |x| File |
23     some (d x containedFiles[sum(size) as totalSize]) where size = totalSize
24
25 // The depth of a file or directory is equal to the depth of its parent + 1
26 forall d : Dir[oid,depth], o : (Dir + File)[oid,depth] |
27   o[oid][oid as to] in (d[oid][oid as from] |x| contents)[to] =>
28     some (o[oid as to] x d[depth as parentDepth]) where (depth = parentDepth + 1)
29
30 // The depth of Root is 0
31 some (Root |x| Dir) where depth = 0
32
33 // Get a solution with the least number of files and directories
34 objectives: minimize (File + Dir)[count()]
```

Listing 1. AlleAlle specification of a small file system, original example comes from [28]. `[..]` is projection, `[.. as ..]` is renaming, `x` is cartesian product, `&` is intersection, `+` is union, $*$ and `^` are (reflexive) transitive closure, `|x|` is natural join.

according to the relation definition, the empty relation is an accepted instance.

The `contents` relation (line 4) is a binary relation encoding which directories and files are contained by some directory. The `..` notation is a short hand notation to define a range of tuples.[3]

Declaring constraints The next part of the specification, lines 6 to 31, describes constraints on the relations. Specifications have no directionality but for clarity of the example we artificially split the constraints in two parts. The first part, lines 6 to 15, defines constraints on the relational shape of the solution. The second part, lines 17 to 31, defines constraints on the data.

Line 7 constrains the `contents` relation to be a subset of the `Dir x (Dir + File)` relation. This enforces the `contents` relation to only contain tuples of which the **`id`** of the `from` attribute exists in the `Dir` relation and the **`id`** of the `to` attribute exists in either the `Dir` or `File` relation. Without this constraint the content relation might contain junk. In other words, it might contain tuples tying non-existing directories to other non-existing directories or files. Since the relation definition does not state anything on how different relations relate to each other these associations must be supplied as extra constraints.

Union compatibility In Codd's relational algebra the union (`+`), intersection (`&`), difference (`-`), subset (**`in`**), and equality (`=`) operators require relations to be *union compatible* with each other. This means that both relations must have the exact same header (both attribute names and associated domains). For instance, on line 11 the constraint that enforces that `Root` 'is a' `Dir` is expressed using the subset (**`in`**) operator. The header of the root operator on the left hand side only contains the single attribute `oid` of the **`id`** domain. The header of the `Dir` relation on the other hand has three attributes (`oid`, `depth` and `size`). Since the subset operator needs the relations to be union compatible we use the projection (`[]`) operator on the `Dir` relation to project the `oid` attribute out of the `Dir` relation resulting in a new relation with only one attribute of the same domain.

Transitive closure Line 9 states that no directory can contain itself expressed with the use of the transitive closure operator. Both the transitive closure (`^`) and reflexive transitive closure ($*$) are special operators in AlleAlle. They are not part of the traditional relational algebra since it is not

[3] The range `<d0>..<d2>` denotes the tuples `<d0>,<d1>,<d2>`. Likewise, the range `<d0,f0>..<d1,f1>` denotes the tuples `<d0,f0>,<d0,f1>,<d1,f0>,<d1,f1>`.

possible to calculate such a transitive closure on relations in general [2].

Since AlleAlle relations are bounded it is possible to implement both operators, albeit with restriction: both operators only operate on binary relations with two attributes of the `id` domain. Line 9 applies the transitive closure over the `contents` relation.

The other constraints in the first part of the specification ensure that all directories and files are reachable from the `Root` directory (line 13), and that files and directories can only be contained by one directory or none, as is the case for the `Root` directory (line 15).

The used multiplicity constraints `lone` and `some` have their standard semantics: `lone` means zero or one tuple in the relation is required, `some` means at least one tuple is required.

Restriction Lines 17 to 31 define data constraints. Line 18 states that all files must have a positive size. To express this constraint the restrict operator (`where`) is used. Using the restrict operator we can formulate constraints on the attributes of a relation. Applying the restrict operator on a relation results in another relation. To enforce that this restriction holds for all files, the multiplicity constraint `some` is used. This forces the restricted relation to contain at least one tuple.

Aggregate functions Lines 21–23 define the value of the size attribute of directories. The size of a directory in the file system is the summation of the sizes of the files which are (transitively) contained by the directory. On line 22 the `containedFiles` relation of the current directory `d` is defined using the transitive closure of the `contents` relation which is (naturally) joined (`|x|`) with the current directory `d`. This relation is then used to calculate the size of the directory by using the aggregation function `sum`.

The `sum` function sums up all the values of the `size` attributes in the `containedFiles` relation. The result of applying an aggregation function is yet another relation containing zero or one tuples with one attribute (in this case `totalSize`). Other available aggregation functions include `min`, `max`, `avg` and `count`. Count is the only aggregation function that does not need an attribute to perform the aggregation on since it counts the number of tuples in the relation. Note that with the use of the count aggregation function all other multiplicity constraints (`some`, `one`, `lone`, `no`) can be expressed.

The remaining constraints describe the value of the `depth` attribute (line 26–28) and enforce the depth of the `Root` directory to be zero (line 31).

Optimization objectives The last line, line 34, defines a single optimization objective. This objective states that we want to optimize on the cardinality of the `File` and `Dir` relations. Only a relation with a single integer attribute can be used as an optimization criterion.

The optimization criteria are so called "soft constraints". This means that, other than the previously described "hard constraints" (lines 6–31), they do not influence the total number of satisfying instances of a problem. They do influence the order in which the model finder returns solutions. In other words, the instance that is returned first will be the instance that is optimal considering the optimization objectives.

$$
\begin{aligned}
File &\mapsto \{\langle f0, 100, 2\rangle\} \\
Dir &\mapsto \{\langle d0, 100, 0\rangle, \langle d1, 100, 1\rangle\} \\
Root &\mapsto \{\langle d0\rangle\} \\
contents &\mapsto \{\langle d0, d1\rangle, \langle d1, f0\rangle\}
\end{aligned}
$$

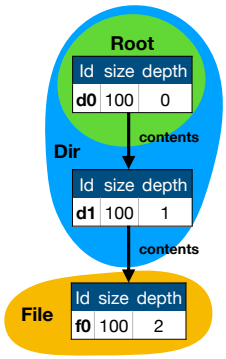

Figure 1. The minimal instance of the small file system specification

The found solution Figure 1 shows the minimal solution of the file system specification. The found solution contains a binding for all the declared relations. Since the optimization objective stated that we wanted a minimal number of files and directories, it returned a solution that contains only those tuples that were part of our lower bound definition (in the case of `File` and `Root`) and those tuples that were needed to get a consistent model according to the described constraints (in case of the `Dir` and `contents` relations).

Next to that, the model finder returned a binding for all the introduced holes. The definition of the `File` tuple `<f0,2,100>` and the constraints on the data determined the values of the other `depth` and `size` attributes.

3 Formal Definition of AlleAlle

Figure 2 shows the abstract syntax of AlleAlle. We define an AlleAlle problem as follows. A **problem** P consists of relation definitions $R_1 \ldots R_n$ which are bound to a relational variables $r_1 \ldots r_n$, formulas $F_1 \ldots F_n$ and possibly optimization criteria $O_1 \ldots O_n$. A **formula** F is a sentence over an alphabet of the relational variables $r_1 \ldots r_n$. A **binding** b is an instance of all the problem's free relational variables to relational constants. A **relational constant** of R is a set of tuples. The binding b is said to be a satisfying instance of P if it conforms to the relation definitions of P and makes all the formulas of P true.

```
problem ::= relDecl* alleForm* objective*
relDecl ::= x ( relHeader ) relBody
relHeader ::= {x : domain}*
relBody ::= = bound | <= bound | >= bound <= bound
bound ::= tupleDecl*
tupleDecl ::= ⟨value*⟩
value ::= x | n | ?
domain ::= id | int

alleForm ::= not alleForm                                   negation
           | no alleExpr                                    empty
           | lone alleExpr                                  at most one
           | one alleExpr                                   exactly one
           | some alleExpr                                  at least one
           | alleExpr in alleExpr                           subset
           | alleExpr = alleExpr                            equal
           | alleForm || alleForm                           disjunction
           | alleForm && alleForm                           conjunction
           | alleForm => alleForm                           implication
           | alleForm <=> alleForm                          equality
           | forall x* : alleExpr | alleForm                universal
           | exists x* : alleExpr | alleForm                existential
           | let x* = alleExpr | alleForm                   let

alleExpr ::= x
           | alleExpr [x* as x*]                            renaming
           | alleExpr [x*]                                  projection
           | alleExpr where condition                       restriction
           | ^alleExpr                                      trans. closure
           | *alleExpr                                      refl. trans. clos.
           | alleExpr [aggFunc*]                            aggregate
           | alleExpr + alleExpr                            union
           | alleExpr & alleExpr                            intersection
           | alleExpr - alleExpr                            difference
           | alleExpr x alleExpr                            product
           | alleExpr |x| alleExpr                          natural join

condition ::= not condition
            | condition && condition
            | condition || condition
            | conditionExpr ( < | <= | > | >= | = ) conditionExpr

conditionExpr ::= x | n | |conditionExpr| | - conditionExpr
                | conditionExpr ( + | - | * | / | % ) conditionExpr

aggFunc ::= count() | sum(x) | min(x) | max(x) | avg(x)      aggr. func.

objective ::= maximize alleExpr| minimize alleExpr          obj. crit.
```

Figure 2. Abstract Syntax of AlleAlle

Relations are defined as in the relational model [10, 12]. A **relation** R over multiple domains $D_1 \dots D_n$, not necessary distinct, consists of a *header* H and a *body* B. The **header** H consists of a fixed set of attribute names, domain pairs $\{\langle a_1 : D_1\rangle \dots \langle a_n : D_n\rangle\}$. An **attribute name** is an arbitrary label. A **domain** is a named set of scalar values, all of the same type. Attribute names in a relation are distinct.

A **body** B consists of a set of *tuples* $\{T_1 \dots T_n\}$. Since the body is a true set it means that per definition the tuples in the body must be unique. A **tuple** T consists of a set of *attribute name*, *value* pairs $\{\langle a_1 : v_1\rangle \dots \langle a_n : v_n\rangle\}$. For each attribute name and domain pair $\langle a_n : D_n\rangle$ in H there exists an attribute name and value pair $\langle a_n : v_n\rangle$ in T where v_n is drawn from D_n. Relations can be bound to relational variables which are arbitrary labels.

3.1 Attribute Domains in AlleAlle

As stated in the definition above, the attribute domains are named sets of scalar values. Currently AlleAlle supports the `int` and `id` domains. The scalars of the `int` domain are defined by the underlying SMT solver which is the unbounded set of all integer numbers.

The `id` domain is a bounded domain consisting of arbitrary chosen labels (atoms). Like mentioned earlier, it contains exactly those atoms that are defined in a specification. Please note that the existence of this domain is not strictly essential (since it could be modeled using `int` values) but it allows for a convenient way to model different dependencies between relations (associations, containment, specialization, etc.).

3.2 Semantics

Figure 3 shows the semantics of AlleAlle. We do not include the semantics of the optimization objectives since they are orthogonal to the semantics of formulas and expressions, and defined in terms of the underlying SMT solver, νZ [7]. The meaning of an AlleAlle problem is defined by four functions, P, R, F and E, which can be recursively applied. The function R accepts a relation declaration and binding and returns whether the header of the binding is equal to the header of the declaration and whether the lower bound of the declaration is a subset of the body of the binding which in turn must be a subset of the upper bounds of the declaration. The function F accepts an AlleAlle formula and a binding and returns whether the binding satisfies the formula. The function E accepts an AlleAlle expression and a binding and returns a relational constant. The function P acts as the starting point and accepts a `Problem` and a binding, and calls the R and F functions.

The semantics of the rename, project, restrict, and aggregate operators on relations are defined in the standard way [10]. The same goes for union ($\cup$), intersection ($\cap$), difference ($\setminus$) and cartesian product ($\times$). Union, intersection and difference can only be applies on union compatible relations (see 2.1). Cartesian product requires two relations with disjoint headers. Applying the cartesian product on two relations with respectively n and m sized tuples *flattens* both relations into a new relation of $n + m$ sized tuples.

The semantics of AlleAlle shown in Figure 3 defines the meaning of an AlleAlle problem given an assignment of relation constants to all relational variables in the binding b. In order to find solutions rather than check their truth

$$
\begin{array}{ll}
P & : problem \rightarrow binding \rightarrow boolean \\
R & : relDecl \rightarrow binding \rightarrow boolean \\
F & : alleForm \rightarrow binding \rightarrow boolean \\
E & : expr \rightarrow binding \rightarrow constant \\
binding & : var \rightarrow constant
\end{array}
$$

$$P[\![r_1 \ldots r_n\ f_1 \ldots f_m]\!]b = R[\![r1]\!]b \wedge \ldots \wedge R[\![r_n]\!]b \wedge F[\![f_1]\!]b \wedge \ldots \wedge F[\![f_m]\!]b$$

$$R[\![x\ (h)\ [l,u]\ ]\!]b = h = b[x]_{header} \wedge l \subseteq b[x]_{body} \subseteq u$$

$$
\begin{array}{ll}
F[\![\texttt{not}\ f]\!]b & = \neg F[\![f]\!]b \\
F[\![\texttt{no}\ r]\!]b & = |E[\![r]\!]b| = 0 \\
F[\![\texttt{lone}\ r]\!]b & = |E[\![r]\!]b| \leq 1 \\
F[\![\texttt{one}\ r]\!]b & = |E[\![r]\!]b| = 1 \\
F[\![\texttt{some}\ r]\!]b & = |E[\![r]\!]b| > 0 \\
F[\![r\ \texttt{in}\ s]\!]b & = E[\![r]\!]b \subseteq E[\![s]\!]b \\
F[\![r = s]\!]b & = E[\![r]\!]b \subseteq E[\![s]\!]b \wedge E[\![s]\!]b \subseteq E[\![r]\!]b \\
F[\![f\ ||\ g]\!]b & = F[\![f]\!]b \vee F[\![g]\!]b \\
F[\![f\ \&\&\ g]\!]b & = F[\![f]\!]b \wedge F[\![g]\!]b \\
F[\![f\ \texttt{=>}\ g]\!]b & = \neg F[\![f]\!]b \vee F[\![g]\!]b \\
F[\![f\ \texttt{<=>}\ g]\!]b & = F[\![f]\!] \iff F[\![g]\!]b \\
F[\![\texttt{forall}\ v_1 : r_1 \ldots v_n : r_n \mid f]\!]b & = \wedge_{t \in E[\![e_1]\!]b} \big(F[\![\texttt{forall}\ v_2 : e_2 \ldots v_n : r_n \mid f]\!](b \oplus v_1 \mapsto \{t\})\big) \\
F[\![\texttt{exists}\ v_1 : r_1 \ldots v_n : r_n \mid f]\!]b & = \vee_{t \in E[\![r_1]\!]b} \big(F[\![\texttt{exists}\ v_2 : r_2 \ldots v_n : r_n \mid f]\!](b \oplus v_1 \mapsto \{t\})\big) \\
F[\![\texttt{let}\ v_1 : r_1 \ldots v_n : r_n \mid f]\!]b & = F[\![\texttt{let}\ v_2 : r_2 \ldots v_n : r_n \mid f]\!](b \oplus v_1 \mapsto E[\![r_1]\!]b)
\end{array}
$$

$$
\begin{array}{ll}
E[\![x]\!]b & = b[x] \\
E[\![r[a_1\ \texttt{as}\ aa_1, \ldots a_n\ \texttt{as}\ aa_n]]\!]b & = \rho_{(aa_1/a_1 \ldots aa_n/a_n)} E[\![r]\!]b \\
E[\![r[a_1 \ldots a_n]]\!]b & = \Pi_{(a_1 \ldots a_n)} E[\![r]\!]b \\
E[\![r\ \texttt{where}\ c]\!]b & = \sigma_c E[\![r]\!]b \\
E[\![\hat{}r]\!]b & = \text{let } m \leftarrow E[\![r]\!]b \text{ in } \langle m_{header}, \{(x : id_x, y : id_y) \mid \exists\, id_1 \ldots id_n \mid \\
 & \quad (x : id_x, y : id_1), (x : id_1, y : id_2) \ldots (x : id_n, y : id_y) \in m_{body}\}\rangle \\
E[\![*r]\!]b & = E[\![\hat{}r]\!]b \cup \mathbb{I} \\
E[\![r[f()]]\!]b & = f()E[\![r]\!]b \text{ (where } f \text{ is } \texttt{count}) \\
E[\![r[f(a)]]\!]b & = f(a)E[\![r]\!]b \text{ (where } f \text{ is } \texttt{sum}, \texttt{avg}, \texttt{min} \text{ or } \texttt{max}) \\
E[\![r + s]\!]b & = E[\![r]\!]b \cup E[\![s]\!]b \\
E[\![r\ \&\ s]\!]b & = E[\![r]\!]b \cap E[\![s]\!]b \\
E[\![r - s]\!]b & = E[\![r]\!]b \setminus E[\![s]\!]b \\
E[\![r\ \texttt{x}\ s]\!]b & = E[\![r]\!]b \times E[\![s]\!]b \\
E[\![r\ |\texttt{x}|\ s]\!]b & = E[\![r]\!]b \bowtie E[\![s]\!]b
\end{array}
$$

Figure 3. Semantics of AlleAlle. Variables f and g range over formulas, r_n and s over expressions. The $\oplus$ operator updates bindings. $\mathbb{I}$ represents the binary identity relation on all values in the `id` domain. $\cup$, $\cap$, $\setminus$ and $\times$ have their standard relational algebra semantics.

$$
\begin{array}{lll}
form & ::= & \top \mid \bot \mid x \mid \neg form \mid form \wedge form \mid form \vee form \mid \\
 & & expr\ (< \mid \leq \mid = \mid \geq \mid >)\ expr \\
expr & ::= & literal \mid x \mid expr + expr \mid expr - expr \mid expr * expr \mid \\
 & & expr\ /\ expr \mid expr\ \%\ expr \mid form\ ?\ expr\ :\ expr
\end{array}
$$

Figure 4. Definition of *form* and *expr*.

value, however, AlleAlle problems are translated to SMT formulas.

4 Translating AlleAlle to SMT

Specifications are translated to SMT constraints. Figure 4 describes the definition of the resulting formula (`form`) that the translation algorithm produces. Our prototype of AlleAlle translates AlleAlle problems to the standard SMT-LIB format, which is supported by multiple SMT solvers [6]. As a result, AlleAlle can potentially be used in combination with different SMT solvers as backends.[4]

Apart from the optimization criteria, the translation consists of flattening AlleAlle problems to a single SMT formula within the logic fragment of quantifier-free non-linear integer arithmetic (QF-NIA). This means that the final SMT formula is a large, but flat formula made up of negation, conjunction, disjunction, integer arithmetic, (in)equalities and if-then-else constructs, as shown in Figure 4.

Before we go into the details of the translation rules we will give an example of how translation unfolds for a small problem.

4.1 Translation Example

Assume we have a relation `Person` with two attributes `pId` and `age` which is defined as follows:

```
Person (pId: id, age: int) <= {<p1,17>,<p2,?>}
```

This relation has an upper bound containing two tuples. The lower bound is omitted and thus empty (i.e. the empty set). Consequently, a satisfying instance may hold zero, one or two tuples in the `Person` relation. The first tuple, `<p1,17>`, assigns the value 17 to the `age` attribute. This means that if this tuple is present the value of `age` must be 17. In the second tuple, `<p2,?>`, the value of the `age` is left open meaning that the value is left to the underlying solver.

Next we define the following constraint:

```
some Person where age >= 18
```

This constraint states that there must be at least one person who is an adult. Or more precisely, there needs to be at least one tuple in the `Person` relation where the value of the age attribute is equal to or greater than 18. Please note that the first tuple in the relation, `<p1,17>`, can never satisfy this constraint since the value of its `age` attribute will always be 17.

As a first step in the translation an environment ρ is constructed, mapping relation names (e.g., `Person`) to an internal relation representation. In the example the created environment ρ is as follows:

$$\rho = \left(\begin{array}{lll|ll} & \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline \texttt{Person} \mapsto & \text{p1} & 17 & b_0 & \top \\ & \text{p2} & i_0 & b_1 & \top \end{array} \right)$$

The internal representation of the `Person` relation consists of a table, with columns for the declared attributes `pId` and `age`, and two additional columns, `exists` and `attCons`. The `pId` attribute contains the values `p1` and `p2` both drawn from the `id` domain. For the first tuple the `age` attribute contains the constant 17. This is a consequence of the given relation

[4] Currently *optimization criteria* are not supported by all SMT solvers. At least Z3 [7] and MathSAT5 [26] have built-in support.

definition where the age attribute for this tuple was assigned 17. For the second tuple the age attribute contains an integer variable i_0. Since in the relation definition this value was left open it is converted to a fresh integer variable.

The exists column encodes whether the tuple should be present in a satisfying instance or not. In this case the value of the exists column for both tuples contains a fresh boolean variable, b_0 and b_1 respectively. This is due to the fact that both tuples are part of the upper bound of the relation but not of the lower bound (since the lower bound of this relation is the empty set). For each satisfying instance the solver will assign truth values to these variables. For instance, if $b_0 = \top$ and $b_1 = \bot$ it means that the tuple <p1,17> is in the Person relation but <p2,?> is not. The attCons column encodes constraints formulated on the attribute values. Initially the attCons attributes have $\top$ assigned.

The next step in the translation is the translation of the constraints. The translation of constraints consists of the recursive application of two translation functions T_F for the translation of AlleAlle formulas and T_E for the translation of AlleAlle expressions. Their full definitions are shown in Figures 8 and 9. The full translation tree for this example would look like:

$$T_F[\texttt{some Person where age >= 18}]\rho$$
$$\Big|$$
$$T_E[\texttt{Person where age >= 18}]\rho$$
$$\Big|$$
$$T_E[\texttt{Person}]\rho$$

We describe the translation of the example in a bottom-up fashion. The first expression that is translated is the lookup of the Person relation from the environment ρ:

$$T_E[\texttt{Person}]\rho = \rho(\texttt{Person}) = \begin{array}{cc|cc} \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline \text{p1} & 17 & b_0 & \top \\ \text{p2} & i_0 & b_1 & \top \end{array}$$

As shown above, the result of the translation function T_E is another relation. Now the outer where expression is translated as follows:

$$T_E[\texttt{Person where age >= 18}]\rho = \begin{array}{cc|cc} \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline \text{p1} & 17 & b_0 & \bot \\ \text{p2} & i_0 & b_1 & i_0 \geq 18 \end{array}$$

The restriction expression (age >= 18) forces additional constraints on the age attribute. In case of the first tuple the age attribute contains the constant 17. Since 17 is less than 18, the value $\bot$ is assigned to the to the attCons attribute. For the second tuple the age attribute was left open which resulted in the introduction of the i_0 variable. For this tuple the constraint $i_0 \geq 18$ is added to the attCons column.

The last step is the translation of the outer formula: $T_F[\texttt{some Person where age >= 18}]\rho$. The T_F function flattens the relation into a flat SMT formula. The translation of the some operator gives the following result:

$$\begin{aligned} & T_F[\texttt{some Person where age >= 18}]\rho \\ &= \bigvee \begin{array}{cc|cc} \text{pId} & \text{age} & \text{exists} & \text{attCons} \\ \hline \text{p1} & 17 & b_0 & \bot \\ \text{p2} & i_0 & b_1 & i_0 \geq 18 \end{array} \\ &= (b_0 \wedge \bot) \vee (b_1 \wedge i_0 \geq 18) \\ &= b_1 \wedge i_0 \geq 18 \end{aligned}$$

The some formula is satisfied if at least one tuple in the relation exists. This is accomplished by translating it to a disjunction of the conjoined exists and attCons columns of the tuples in the relation (i.e. this is depicted by the big vee notation in the above translation).

As can be seen, the translation of this formula results in the SMT formula $b_1 \wedge i_0 \geq 18$. This means that an instance of this problem is satisfying iff it contains the second tuple (i.e. b_1 must be true) and the value of its age attribute is greater than or equal to 18. The presence or absence of the first tuple does not change the validity of the resulting instance since it assigned age value of 17 never conforms to the formulated constraint. This means that a satisfying instance can either contain or not contain the first tuple as long as the second tuple is present.

$$\begin{array}{lll} \textit{rel} & ::= & \langle \overline{x : \textit{domain}}_{\textit{header}}, \overline{\textit{tuple}}_{\textit{body}} \rangle \\ \textit{tuple} & ::= & \langle \overline{(x_{\textit{name}} : \textit{cell}_{\textit{value}})}_{\textit{attributes}}, \textit{form}_{\textit{exists}}, \textit{form}_{\textit{attCons}} \rangle \\ \textit{cell} & ::= & \textit{atom} \mid \textit{expr} \\ \textit{domain} & ::= & \text{ID} \mid \text{INT} \end{array}$$

Figure 5. Definition of *rel* as used in the translation. expr and form are defined in Figure 4.

4.2 The Algorithm

The translation of an AlleAlle specification starts with the translation of a problem using the function T_P:

$$T_P : \textit{problem} \rightarrow \textit{form}$$
$$T_P[r_1(h_1)b_1 \ldots r_n(h_n)b_n\ f_1 \ldots f_m] = \bigwedge_{i=1}^{m} T_F[f_i]\rho$$
$$\text{where } \rho = \bigcup_{j=1}^{n} \left(r_j \mapsto T_R[(h_j)\ b_j]\right)$$

This function translates the constraints $f_1 \ldots f_m$ of the problem to formulas (of type *form*, see Figure 4). The environment is populated using the T_R function which converts relation declarations to the internal tabular representation. The T_P function returns a conjunction of all the translated constraints.

The Relation data structure Central to the translation is the internal relation data structure, shown in Figure 5, which in this paper we visualize using the tabular notation introduced above. A relation *rel* consists of a header and

$$
\begin{array}{ll}
T_R & : relHeader \rightarrow relBody \rightarrow rel \\
T_R[(h) = b] & = \text{add}(\langle h, \varnothing \rangle, b, \lambda t.\ \langle \text{convert}(t), \top, \top \rangle) \\
T_R[(h) <= ub] & = \text{add}(\langle h, \varnothing \rangle, ub, \lambda t.\ \langle \text{convert}(t), x, \top \rangle) \\
T_R[(h) >= lb <= ub] & = \text{add}(\langle h, \varnothing \rangle, ub, \lambda t.\ \langle \text{convert}(t), \text{exists}(t, lb), \top \rangle)
\end{array}
$$

$$
\begin{array}{ll}
\text{add} & : rel \rightarrow \overline{tupleDecl} \rightarrow (tupleDecl \rightarrow tuple) \rightarrow rel \\
\text{add}[r, b, f] & = \text{if } b = \varnothing \text{ then } r \text{ else let } t \in b \text{ in add}(\text{addDistinct}(r, f(t)), b \setminus t)
\end{array}
$$

$$
\begin{array}{ll}
\text{convert} & : tupleDecl \rightarrow \overline{x : cell} \\
\text{convert}[t] & = \langle (atr_{name} : \begin{cases} id \text{ when } atr = id \\ i \text{ when } atr = hole \text{ (} i \text{ as fresh int var)} \\ constant \text{ when } atr = constant \end{cases} \quad \mid atr \in t) \rangle
\end{array}
$$

$$
\begin{array}{ll}
\text{exists} & : tupleDecl \rightarrow bound \rightarrow form \\
\text{exists}[t, lb] & = \begin{cases} \top \text{ when } t \in lb \\ x \text{ otherwise (with } x \text{ as fresh bool var)} \end{cases}
\end{array}
$$

$$
\text{addDistinct} \quad : rel \rightarrow tuple \rightarrow rel
$$

Figure 6. Definition and construction of relations. The definition of the *addDistinct* function is included in Appendix A. *relHeader*, *relBody*, *bound*, *tupleDecl*, *id* and *hole* are declared in Figure 2. The other definitions are given in Figure 4 and Figure 5.

`File` $\mapsto$

oId	depth	size	exists	attCons
f0	2	100	$\top$	$\top$
f1	i_0	i_1	b_0	$\top$
f2	i_2	i_3	b_1	$\top$

Figure 7. The visual representation of `File rel` after its construction based on the relation declaration: `File (oId:id,depth:int,size:int) >= {<f0,2,100>} <= {<f0,2,100>,<f1,?,?>,<f2,?,?>}`.

a body. The header is defined as a mapping from attribute names to domains. The body is a set of tuples. Each tuple in the body contains the declared `attributes` and two additional columns, `exists` and `attCons`. When we refer to *tuple* we refer to the combination of the `attributes` and `exists` and `attCons` columns. In the translation rules we will use the subscripts r_{header} and r_{body} for the header and body of a relation r and $t_{attributes}$, t_{exists} and $t_{attCons}$ for the `attributes`, `exists` and `attCons` columns of a tuple t to refer to the specific parts of a relation or tuple (see Figure 5).

4.2.1 Constructing relations

A `rel` can be constructed in three different ways depending on how it is declared. Figure 6 shows the definition of the construction function T_R. This function translates the relation definition (i.e. the header and lower and upper bounds) to a `rel`. Which construction function is used depends on how the bounds are declared and influences the value of the tuple's `exists` field. This value depends on whether the tuple declaration is part of the lower and upper bound or only of the upper bound. For instance, in the example of the small file system (Figure 1) the `File` relation is declared with both a lower and an upper bound (e.g. `>= {<f0,2,100>} <= {<f0,2,100>,<f1,?,?>,<f2,?,?>}`).

On construction of the `rel` the value $\top$ is assigned to the `exists` field of the tuple `<f0,2,100>` since there cannot be a satisfying instance without this tuple present. It is a different case for the tuples `<f1,?,?>` and `<f2,?,?>`. Since they are only part of the relation's upper bound there may be satisfying instances where these tuples (or one of these tuples) are not present. To encode this, fresh boolean variables are assigned to the `exists` fields of both tuples. It is then up to the underlying SMT solver to find a satisfying assignment for these boolean variables. The full encoding of the `File` relation is shown in Figure 7 (with the `exists` column highlighted). The encoding is adapted from the relational model finder Kodkod, with the difference that it is encoded in our relation data structure instead of a boolean matrix as used by Kodkod [31].

Ensuring tuple distinctness The `attCons` column holds the constraints that were added on the tuple's scalar attributes. On construction this column will be populated with $\top$ for most tuples. The only exception to this is when the added tuples in the relation are potentially non-distinct. Consider for instance the (valid) case that the `File` relation would have an upper bound of two possible tuples: `<f1,1,10>,<f1,?,?>`. Since they both share the same `oid` value (namely `f1`) these tuples could potentially overlap if the `depth` and `size` attributes of the second tuple would evaluate to 1 and 10 respectively.

Since the relational model dictates true set semantics for its relational bodies, the translation must enforce that all tuples in the relation are distinct, or collapsed into each other. To enforce this the translation algorithm adds a constraint to the `attCons` field of the second tuple that forces the value of either the `depth` or `size` attribute to be different, if the first tuple exists in the relation.

This distinctness rule is added on construction of the relations by applying the *addDistinct* function (included in Appendix A). This function checks whether tuples can potentially overlap and adds the necessary constraints to the `attCons` field. In the case of the example given in this section the constructed relation would be as follows:

	oid	depth	size	exists	attCons
`File` $\mapsto$	f1	1	10	b_0	$\top$
	f1	i_0	i_1	b_1	$\neg b_0 \vee \neg(i_0 = 1 \wedge i_1 = 10)$

4.2.2 Translating AlleAlle constraints

The entry for translating AlleAlle formulas to SMT formulas is the T_F function, shown in Figure 8. To translate an AlleAlle expression the T_F function calls the T_E function which is defined in Figure 9.

$env : identifier \to rel$

$T_F : alleForm \to env \to form$
$T_F[\texttt{not}\ f]\rho = \neg T_F[f]\rho$
$T_F[\texttt{no}\ r]\rho = \neg T_F[\texttt{some}\ r]\rho$
$T_F[\texttt{lone}\ r]\rho = T_F[\texttt{no}\ r]\rho \vee T_F[\texttt{one}\ r]\rho$
$T_F[\texttt{one}\ r]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in } \bigvee_{t \in m_{body}} \Big(\text{tg}(t) \wedge \big(\bigwedge_{t' \in m_{body}, t' \neq t} (\neg \text{tg}(t'))\big)\Big)$
$T_F[\texttt{some}\ r]\rho = \text{let } m \leftarrow T_E[p]\rho \text{ in } \bigvee_{t \in m_{body}} \big(\text{tg}(t)\big)$
$T_F[r\ \texttt{in}\ s]\rho = \text{let } m \leftarrow T_E[r]\rho,\ \text{let } n \leftarrow T_E[s]\rho$
$\Big(\bigwedge_{t \in m_{body}, u \in n_{body}, \text{canOverlap}(t,u)} (\neg tg(t) \vee (tg(u) \wedge \text{attEqual}(t,u)))\Big) \wedge$
$\Big(\bigwedge_{t \in m_{body}, t \notin n_{body}} \neg tg(t)\Big)$
$T_F[r = s]\rho = T_F[r\ \texttt{in}\ s]\rho \wedge T_F[s\ \texttt{in}\ r]\rho$
$T_F[f\ \texttt{||}\ g]\rho = T_F[f]\rho \vee T_F[g]\rho$
$T_F[f\ \texttt{\&\&}\ g]\rho = T_F[f]\rho \wedge T_F[g]\rho$
$T_F[f\ \texttt{=>}\ g]\rho = \neg T_F[f]\rho \vee T_F[g]\rho$
$T_F[f\ \texttt{<=>}\ g]\rho = T_F[f\ \texttt{=>}\ g]\rho \wedge T_F[g\ \texttt{=>}\ f]\rho$
$T_F[\texttt{forall}\ v_1 : ex_1 \ldots v_n : ex_n \mid f]\rho = \text{let } m \leftarrow T_E[ex_1]\rho \text{ in}$
$\bigwedge_{t \in m_{body}} \Big(\neg tg(t) \vee T_F[\texttt{forall}\ v_2 : ex_2 \ldots v_n : ex_n \mid f]\rho\big[v_1 \mapsto \text{sing}(m_{header}, t)\big]\Big)$
$T_F[\texttt{exists}\ v_1 : ex_1 \ldots v_n : ex_n \mid f]\rho = \text{let } m \leftarrow T_E[ex_1]\rho \text{ in}$
$\bigvee_{t \in m_{body}} \Big(tg(t) \wedge T_F[\texttt{exists}\ v_2 : ex_2 \ldots v_n : ex_n \mid f]\rho\big[v_1 \mapsto \text{sing}(m_{header}, t)\big]\Big)$
$T_F[\texttt{let}\ v_1 : ex_1 \ldots v_n : ex_n \mid f]\rho = T_F[\texttt{let}\ v_2 : ex_2 \ldots v_n : ex_n \mid f](\rho[v_1 \mapsto T_E[ex_1]\rho]$

$\text{tg} : tuple \to form$
$\text{tg}[t] = t_{exists} \wedge t_{attCons}$

$\text{sing} : \overline{x : domain} \to tuple \to rel$
$\text{sing}[h, t] = \big\langle h, \{\langle t_{attributes}, \top, t_{attCons}\rangle\}\big\rangle$

Figure 8. Translation rules for AlleAlle formulas. r and s are *alleExpr*, f and g are *alleForm*. Definitions of *canOverlap* and *attEqual* are included in Appendix A. *tg* is short for *together* and *sing* is short for *singleton* meaning a relation with only one tuple.

The T_E function translates the expression and returns a new `rel`. The T_F function flattens the translated `rels` into SMT formulas. These in turn get conjoined by the T_P function introduced earlier.

Tuple equality constraints When translating the *subset* formula and the *union*, *intersection* and *difference* expressions we again have to account for possible overlapping tuples described earlier. The difference being that in the case of translating the above rules we need to *enforce* equality instead of preventing it. In this case, the constraints that need to be added to the `attCons` field are constraints that force the value of the attributes to be the same; this is done using the helper functions *canOverlap* and *attEquals*, which are included in Appendix A.

As an example consider the following case. Suppose we have the following specification:

```
Shape (sid:id, size:int)  <= {<s1,?>}
Square (sid:id, size:int) <= {<s1,?>}

Square in Shape
```

$T_E : alleExpr \to env \to rel$
$T_E[v]\rho = \rho(v)$
$T_E\Big[r[a_1\ \texttt{as}\ a'_1 \ldots a_n\ \texttt{as}\ a'_n]\Big]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in}$
$T_E\Big[\langle m_h[a_1/a'_1], m_b[a_1/a'_1]\rangle[a_2\ \texttt{as}\ a'_2\ \ldots a_n\ \texttt{as}\ a'_n]\Big]\rho$
$T_E\Big[r[a_1 \ldots a_n]\Big]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in}$
$\langle (a : m_h[a] \mid a \in a_1 \ldots a_n), \text{merge}(m_b, a_1 \ldots a_n)\rangle$
$T_E[r\ \texttt{where}\ c]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in } \big\langle m_h, \{\langle t_a, t_e, t_a \wedge T_C[c]t\rangle \mid t \in m_b\}\big\rangle$
$T_E[\hat{}r]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in}$
$\text{let sqr} \leftarrow \lambda r.i.\ \text{if } i >= |m_{body}| \text{ then } r \text{ else joinOnce}(r, i * 2) \text{ in sqr}(m, 1)$
$T_E[*r]\rho = T_E[\texttt{iden}\ +\ \hat{}r]\rho$

$T_E\Big[r[f(a)\ \texttt{as}\ x]\Big]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in (where } f \text{ is } \texttt{count}, \texttt{sum}, \texttt{avg})$
$\Big\langle (x : \texttt{int}), \langle (x : i), \top, i = T_{Ag}[f(a)]m_b\rangle\Big\rangle$
$T_E\Big[r[f(a)\ \texttt{as}\ x]\Big]\rho = \text{let } m \leftarrow T_E[r]\rho \text{ in (where } f \text{ is } \texttt{min}, \texttt{max})$
$\Big\langle (x : \texttt{int}), \langle (x : i), \text{if } |m| > 0 \text{ then } \top \text{ else } \bot, i = T_{Ag}[f(a)]m_b\rangle\Big\rangle$

$T_E[r + s]\rho = \text{let } m \leftarrow T_E[r]\rho,\ \text{let } n \leftarrow T_E[s]\rho \text{ in}$
$\Big\langle m_h, \{\langle t_a, t_e \vee u_e, t_c \wedge u_c \wedge \text{attEqual}(t,u)\rangle \mid t \in m_b, u \in n_b, \text{canOverlap}(t,u)$
$\cup (m_b \setminus n_b) \cup (n_b \setminus m_b)\}\Big\rangle$
$T_E[r\ \&\ s]\rho = \text{let } m \leftarrow T_E[r]\rho,\ \text{let } n \leftarrow T_E[s]\rho \text{ in}$
$\Big\langle m_h, \{\langle t_a, t_e \wedge u_e, t_c \wedge u_c \wedge \text{attEqual}(t,u)\rangle \mid t \in m_b, u \in n_b, \text{canOverlap}(t,u)\}\Big\rangle$
$T_E[p - q]e = \text{let } m \leftarrow T_E[p]\rho,\ \text{let } n \leftarrow T_E[q]\rho \text{ in}$
$\Big\langle m_h, \{\langle t_a, t_e \wedge \neg u_e, t_c \wedge \text{attEqual}(t,u)\rangle \mid t \in m_b, u \in n_b, \text{canOverlap}(t,u)$
$\cup (m_b \setminus n_b)\}\Big\rangle$
$T_E[r \times s]\rho = \text{let } m \leftarrow T_E[r]\rho,\ \text{let } n \leftarrow T_E[s]\rho \text{ in}$
$\Big\langle m_h \cup n_h, \{\langle t_a \cup u_a, t_e \wedge u_e, t_c \wedge u_c\rangle \mid t \in m_b, u \in n_b, t_a \cup u_a = \varnothing\}\Big\rangle$
$T_E[p\ |\times|\ q]\rho = \text{let } m \leftarrow T_E[p]\rho,\ \text{let } n \leftarrow T_E[q]\rho \text{ in}$
$\Big\langle m_h \cup n_h, \{\langle t_a \cup u_a, t_e \wedge u_e, t_c \wedge u_c\rangle \mid t \in m_b, u \in n_b, t_a \cup u_a \neq \varnothing\}\Big\rangle$

$\text{merge} : \overline{tuple} \to \overline{x} \to \overline{tuple}$
$\text{joinOnce} : rel \to x \to y \to rel$

Figure 9. Translation rules for AlleAlle expressions. For abbreviation purposes the notation r_h means r_{header}, r_b means r_{body}, t_a means $t_{attributes}$, t_e means t_{exists} and t_c means $t_{attCons}$. r and s are *alleExpr*, c is a *condition*. Definitions of *canOverlap* and *attEqual* are included in Appendix A. The definition of *merge* and *joinOnce* is not given but sketched in the text. `iden` resolves to the binary identity relation on all values in the `id` domain.

We define two relations, `Shape` and `Square`. Both relations have an `sid` field of type `id` and a `size` field of type `int`. The `sid` fields contain the same `id` literal and both `size` attributes have been left open. Translating the constraint `Square in Shape` would yield the following result:

$T_E[\texttt{Shape}]\rho =$

sid	size	exists	attCons
s1	i_0	b_0	$\top$

$T_E[\texttt{Square}]\rho =$

sid	size	exists	attCons
s1	i_1	b_1	$\top$

$T_F[\texttt{Square in Shape}]\rho = \neg b_1 \vee (b_0 \wedge i_0 = i_1)$

$$
\begin{aligned}
&T_C && : tuple \rightarrow form \\
&T_C[\![!c]\!]t && = \neg T_C[\![c]\!]t \\
&T_C[\![c_1 \texttt{ \&\& } c_2]\!]t && = T_C[\![c_1]\!]t \wedge T_C[\![c_2]\!]t \\
&T_C[\![c_1 \texttt{ || } c_2]\!]t && = T_C[\![c_1]\!]t \vee T_C[\![c_2]\!]t \\
&T_C[\![e_1 \odot e_2]\!]t && = T_{Ce}[\![e_1]\!]t \odot T_{Ce}[\![e_2]\!]t
\end{aligned}
$$

$$
\begin{aligned}
&T_{Ce} && : tuple \rightarrow expr \\
&T_{Ce}[\![x]\!]t && = t_{attributes}[x] \\
&T_{Ce}[\![n]\!]t && = n \\
&T_{Ce}[\![- e]\!]t && = -T_{Ce}[\![e]\!]t \\
&T_{Ce}[\![|e|]\!]t && = \text{let } i \leftarrow T_{Ce}[\![e]\!]t \text{ in } i < 0 \ ? \ -i \ : \ i \\
&T_{Ce}[\![e_1 \oplus e_2]\!]t && = T_{Ce}[\![e_1]\!] \oplus T_{Ce}[\![e_2]\!]
\end{aligned}
$$

Figure 10. Translation rules for the restriction conditions. $\odot$ depicts the different equality operators (`<`, `<=`, `=`, `=>`, `>`), $\oplus$ depicts the arithmetic operators (`+`, `-`, `*`, `/`, `%`).

The outcome is that a `Square` can only be a `Shape` if either the `Square` relation is empty (by enforcing that $b_1 = \bot$) or the value of the `size` attributes of both relations is equal (by enforcing that $i_0 = i_1$). Otherwise the tuple in `Square` would not overlap with the tuple in `Shape` and thus would not be in the *subset* relation.

Translation of the projection expression Projection can reduce the numbers of the tuples in the relation by truncating it. This again can potentially cause tuple overlap. Consider for instance the `Person` relation introduced in the translation example (Section 4.1). This relation has two attributes, `pId` and `age` and was defined as follows:

```
Person (pId:id, age:int) <= {<p1,17>,<p2,?>}.
```

resulting in the following internal representation:

	pId	age	exists	attCons
Person ↦	p1	17	b_0	$\top$
	p2	i_0	b_1	$\top$

If we would project the `age` attribute both tuples could potentially collapse into each other. This would be the case if both tuples exist and the value of the `age` attribute of the second tuple would also be 17. To prevent this the *merge* function adds extra constraints to the tuples reusing the *addDistinct* function (see Appendix A) enforcing that in all possible evaluations of its variables the result would be a relation with distinct tuples. We would end up with a relation with the following values and constraints:

	age	exists	attCons
Person ↦	17	b_0	$\top$
	i_0	b_1	$\neg b_0 \vee \neg(i_0 = 17)$

Translation of the aggregation expression Aggregation results in a new relation containing zero or one tuple with a single attribute. The value of the `attCons` field of this tuple contains the unfolded aggregation expression. The translation of the different rules are shown in Figure 11.

$$
\begin{aligned}
&T_{Ag} : aggFunc \rightarrow \overline{tuple} \rightarrow form \\
&T_{Ag}[\![\texttt{count}()]\!]b = \text{let cnt} \leftarrow \lambda b'.term. \text{ if } b' = \varnothing \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } term + \text{cnt}(b' \setminus t, \text{tg}(t) \ ? \ 1 \ : \ 0) \text{ in cnt}(b, 0) \\
&T_{Ag}[\![\texttt{sum}(a)]\!]b = \text{let sum} \leftarrow \lambda b'.term. \text{ if } b' = \varnothing \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } term + \text{sum}(b' \setminus t, \text{tg}(t) \ ? \ t_{attributes}[a] \ : \ 0) \text{ in sum}(b, 0) \\
&T_{Ag}[\![\texttt{avg}(a)]\!]b = T_{Ag}[\![\texttt{sum}(a)]\!]b \div T_{Ag}[\![\texttt{count}()]\!]b \\
&T_{Ag}[\![\texttt{min}(a)]\!]b = \text{let min} \leftarrow \lambda b'.term. \text{ if } b' = \varnothing \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } t_{attributes}[a] < term \wedge \text{tg}(t) \ ? \\
&\quad\quad \min(b' \setminus t, t_{attributes}[a]) : \min(b' \setminus t, term) \text{ in } \min(b, \text{findFirst}(a, b)) \\
&T_{Ag}[\![\texttt{max}(a)]\!]b = \text{let max} \leftarrow \lambda b'.term. \text{ if } b' = \varnothing \text{ then } term \text{ else} \\
&\quad \text{let } t \in b' \text{ in } t_{attributes}[a] > term \wedge \text{tg}(t) \ ? \\
&\quad\quad \max(b' \setminus t, t_{attributes}[a]) : \max(b' \setminus t, term) \text{ in } \max(b, \text{findFirst}(a, b))
\end{aligned}
$$

$$
\begin{aligned}
&\text{findFirst} : x \rightarrow \overline{tuple} \rightarrow expr \\
&\text{findFirst}[a, ts] = \text{if } ts = \varnothing \text{ then } 0 \text{ else let } t \in ts \text{ in } t_{attributes}[a]
\end{aligned}
$$

Figure 11. Translation rules for the aggregation functions

In case of the application of the `count`, `sum` and `avg` aggregation the resulting relation will always contain a single tuple, even if the aggregated relation was empty. The intuition behind this is that even if the aggregated relation is empty its cardinality is zero and the sum of one of its attributes will also result in zero. The resulting relation after applying the `min` and `max` aggregation could be empty since calculating the `max` or `min` of an empty relation is undefined.

Translation of the transitive closure expression Transitive closure is only defined for binary relations containing two `id` attributes. In theory it is possible to define transitive closure for other data domains but the existence of possible holes would complicate the generation of the right equality constraints. Because of this reason we decided to postpone the calculation of transitive closure for other domain to possible future work.

In essence, calculating the transitive closure can be performed by recursively joining the relation with itself. To calculate the transitive closure we apply a variant of *iterative squaring* that works on our relation data structure. On every iteration the same translations are applied to the previously calculated relation. A single step of this translation is applied in the declared but not defined *joinOnce* function.

4.3 Testing the Translation

To gain more confidence in the correctness of the AlleAlle translation we compared AlleAlle with Kodkod [28] and the Choco solver [23] on a number of different Constraint Satisfaction Problems (CSP) an Constraint Optimization Problems (COP). Kodkod is also a relational model finder but is not able to solve optimization problems directly. The Choco solver is able to solve optimization problems but its formalism and solving strategy is semantically further away from AlleAlle.

Table 1. Testing AlleAlle against Kodkod and Choco.

Problem	Problem type	Compare with	Sat. AlleAlle?	Sat. other?	#inst AlleAlle	#inst other	Same opt. solution?
FileSystem	CSP	Kodkod	Yes	Yes	2184	2184	-
Handshake	CSP	Kodkod	Yes	Yes	24	24	-
Pigeonhole	CSP	Kodkod	No	No	-	-	-
RingElection	CSP	Kodkod	Yes	Yes	2	2	-
RiverCrossing	CSP	Kodkod	Yes	Yes	2	2	-
8Queens	CSP	Choco	Yes	Yes	92	92	-
Sudoku	CSP	Choco	Yes	Yes	1	1	-
SendMoreMoney	CSP	Choco	Yes	Yes	1	1	-
Knapsack	COP	Choco	Yes	Yes	-	-	Yes
Mariokart	COP	Choco	Yes	Yes	-	-	Yes

In the comparison of CSP problems we compare whether AlleAlle and Kodkod, or AlleAlle and Choco find the same answer, and whether they produce the same number of satisfying instances. When comparing COP problems we check whether both AlleAlle and Choco find the same optimal solution. All problems are existing examples or benchmark problems from Kodkod or Choco[5]. The results are shown in Table 1.[6]

As can be seen in the results AlleAlle finds the same solutions as Kodkod and Choco for all implemented problems. Please note that the reported found instances also contain all symmetric solutions. For instance the 8 queens problem has 92 solutions, but if symmetry is taken into account only 12 distinct solutions remain. Kodkod can detect such symmetries by generating so called *symmetry breaking predicates* but for this benchmark we configured Kodkod not to do this [28].

5 Evaluation

We evaluate AlleAlle in terms of performance and expressiveness, by comparing the translation and solving times of AlleAlle with Kodkod [28] on different problems, and by implementing a real-world use case, optimal dependency resolution [1], respectively. Finally, we qualitatively compare AlleAlle to similar systems for solving constraint problems.

5.1 Translation and Solving Time Benchmark

We compare AlleAlle's translation and solving time performance against the translation and solving time performance of Kodkod by translating and solving six different problems. Table 2 characterizes the benchmark problems in terms of the kinds of constraints that are used.

For five of these problems we measure the performance for different configurations of the same problem to get insight in the effect of the size of a problem specification. Configuration parameters are the number of atoms or tuples that are allowed to populate the relations, and the allowed bit-width for the integer encoding used for Kodkod.

[5]See https://github.com/chocoteam/samples/

[6]See https://github.com/joukestoel/allealle-benchmark/ for the encoding of the problems.

Table 2. Overview of the benchmarked problems.

Problem	Constraint types
Alloy FileSystem	Relational
Halmos handshake	Relational, cardinality
Pigeonhole	Relational
River crossing puzzle	Relational
Square $y = x^2$	Integer
Account state transition system	Relational, integer

The benchmarks are run on a early 2015 MacBook Pro with a 2,7 GHz quad core Intel i5 processor with 8GB of DDR3 RAM. Java 8 (version 1.8.0_131 by Oracle) is used for all benchmarks. The translation and solving per configuration per problem was run 30 times with a warmup of 10 runs. All caches were flushed between each run. We report the median of the translation and solving times.

AlleAlle is implemented in Rascal [17]. Rascal is a functional programming language designed for the development and analysis of programming languages. It is an interpreted language that runs on the JVM. All AlleAlle benchmarks were run using version 0.12.0.201901101505 of Rascal and version 4.8.0 of Z3.

Kodkod is implemented in Java and was built and run using the previously mentioned Java version. Some of the benchmarked problems contain cardinality and integer constraints. To avoid wrap-around semantics for integer constraints in Kodkod we use Kodkod* [20, 21], which is currently packaged with Alloy 4.2; the solver is configured with SAT4J (version 2.3.5.v20130525). As earlier, we configured Kodkod to not generate symmetry breaking predicates.

Interpreting the results Table 3 contains the results of benchmarking AlleAlle against Kodkod, comparing translation times and solving times. As can be seen in the results, AlleAlle is slower in translating purely relational problems (e.g., FileSystem, Pigeonhole and Rivercrossing). The reason for the slow-down is twofold. First, our current implementation of AlleAlle is a prototype, built as a proof-of-concept to demonstrate the correctness of the translation algorithm. Furthermore, AlleAlle is implemented in Rascal, which is an interpreted language for language prototyping, whereas Kodkod is implemented in Java. We are currently working on a Java implementation of AlleAlle which, we expect, will bring the translation performance up to par with Kodkod since AlleAlle translation algorithms are of the same complexity as Kodkod's translation.

AlleAlle is also slower in solving purely relational problems compared to Kodkod. This can be explained from the fact that AlleAlle generates more clauses than Kodkod. For instance, for the FileSystem problem in the configuration with 30 atoms AlleAlle generates a total of 5 359 296 clauses while Kodkod merely generates 24 753 clauses. The reason for this difference is that Kodkod implements a clause

Table 3. Benchmark comparison between AlleAlle and Kodkod. Comparison shows six problems of which five are shown with different relation sizes, either in number of atoms or integer bit width used.

				AlleAlle				Kodkod			
Problem	#Atoms	Bit width (Kodkod only)	Sat?	Trans. time (in ms)	Solve time (in ms)	#Vars	#Clauses	Trans. time (in ms)	Solve time (in ms)	#Vars	#Clauses
FileSystem^{r}	15	-	SAT	725	20	130	21 839	12	4	135	3235
FileSystem^{r}	30	-	SAT	14 567	110	460	5 359 296	29	8	470	24 753
HandShake^{r,c}	10	4	SAT	480	10	184	1633	12	61	200	9292
HandShake^{r,c}	17	5	UNSAT	2235	69 865	548	6295	23	136 465	578	39 214
Pigeonhole^{r}	9	-	UNSAT	51	10	20	302	2	1	20	137
Pigeonhole^{r}	17	-	UNSAT	77	1180	72	1424	1	45	72	565
Rivercrossing^{r}	12	-	SAT	327	10	74	1621	7	0	68	749
Square^{i}	2	4	SAT	5	10	2	6	2	4	36	3025
Square^{i}	2	10	SAT	5	10	2	6	198	11 883	2052	371 722
Account^{r,i}	12	5	SAT	72	10	33	289	21	79	351	19 698
Account^{r,i}	12	9	SAT	71	10	33	289	460	34 080	5151	480 198

Problem contains r) relational constraints, c) cardinality constraints, i) integer constraints

rewriting system that is much more aggressive than what is currently implemented in AlleAlle [28].

The results show, however, that AlleAlle's native handling of data for problems that contain both relational and integer constraints, pays off, both in translation times and solving times. As mentioned before, Kodkod needs to specifically encode the possible integers up to the configured bit-width. This is needed because it needs to encode integer constraints as part of the SAT formula so that the underlying SAT-solver can solve the problem. This results in more clauses in the generated SAT formula and thus higher translation and solving times. Since AlleAlle does not require this explicit encoding but can use the solvers built-in reasoning power on different theories it does not suffer from the same performance penalty. Therefore its translation and solving times is consistent for the same problem even if larger integer values are required.

For problems which encode explicit cardinality constraints like the HandShake problem we also see better performance of AlleAlle since relations with higher cardinality require a larger bit-width in Kodkod.[7]

5.2 Optimal Dependency Resolution

To evaluate the expressiveness of AlleAlle we have implemented a solution to the optimal package resolution problem, which is common in package managers like npm, apt, or maven [1]. Figure 12 shows an example of a package resolution problem taken from Tucker et.al. [32]. The user asks to install package a; the package resolver needs to compute which packages to install or uninstall in such a way that all dependencies are satisfied and no conflicts are violated. This has been shown to be an NP-complete problem [11].

[7]Explicit meaning using Kodkod's integer cast expression `sum()` on relations and using integer arithmetic expressions to formulate cardinality constraints.

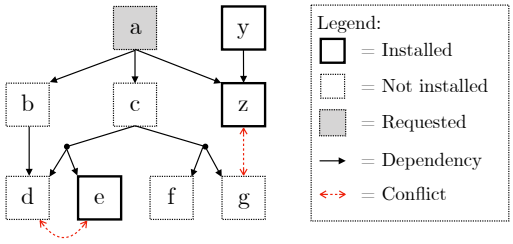

Figure 12. A package resolution problem (solution: install $\{a, b, c, d, f\}$ and uninstall $\{e\}$)

We ran the experiment on the "paranoid" track of the MISC 2012 competition; a solver competition for package managers.[8] This track required the contestants to find the minimal 'change' needed to the system to comply to the package update request.[9] Because this problem requires to reason about the minimal change to the system it can not be directly encoded in Alloy or Kodkod since these formalisms do not allow for the encoding of optimization problems.

All benchmarks were run on the same early 2015 MacBook Pro with a 2,7 GHz quad core Intel i5 processor with 8GB of DDR3 RAM. We compare our found results against the reported solutions and solving times from the MISC 2012 competition.

The package resolution problem can be compactly defined as a relational problem with data. The relevant relations are summarized in Table 4. The first 9 relations represent known facts about the package repository, what is installed on the user's system, and what the user requests to install, remove, or upgrade, respectively. These relations have exact bounds meaning that for every possible solution the tuples

[8]http://www.mancoosi.org/misc-2012/

[9]Minimal change meaning, minimal amount of packages that need to be removed and the minimal amount of packages needed to be installed or updated.

```
// All packages that are requested to be installed or upgraded should be part of the installation afterwards
let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) |
  (forall ir : installRequest | some (ir |x| version |x| installedAfter)
    where ((relop = 0) || (relop = 1 && version = nr) || (relop = 2 && version != nr) || (relop = 3 && nr >= version)|| (relop = 4 && nr <= version)))
  &&
  (forall ur : upgradeRequest | some (ur |x| version |x| installedAfter)
    where ((relop = 0) || (relop = 1 && version = nr) || (relop = 2 && version != nr) || (relop = 3 && nr >= version)|| (relop = 4 && nr <= version)))

forall rr : removeRequest | some (rr |x| version |x| toBeRemovedVersion) // all the removal requests should be scheduled for removal
  where ((relop = 0) || (relop = 1 && version = nr) || (relop = 2 && version != nr) || (relop = 3 && nr >= version) || (relop = 4 && nr <= version))

let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) | // installing version means installing its dependencies afterwards
  forall d : depends | (d[vId] in installedAfter) =>
    let possibleInstalls = ((d |x| dependChoice)[pId,version,relop] |x| (version |x| installedAfter)) |
      (some (possibleInstalls where ((relop = 0) ||(relop = 1 && nr = version)) || (relop = 2 && nr != version) ||
                                     (relop = 3 && nr >= version) ||(relop = 4 && nr <= version))[vId] & installedAfter)

let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) | // when a version is installed, no conflicting version can be installed
  forall c : conflicts | (c[vId] in installedAfter) =>
    let possibleConflicts = (c[pId,version,relop] |x| (version |x| installedAfter)) |
      no (possibleConflicts where ((relop = 0) || (relop = 1 && nr = version) || (relop = 2 && nr != version) ||
                                   (relop = 3 && nr >= version) || (relop = 4 && nr <= version))[vId] & installedAfter)

let installedAfter = (toBeInstalled + (installed - toBeRemovedVersion)) | // all versions to be kept need to be installed afterwards as well
  forall k : keep | some k |x| installedAfter

toBeRemovedPackage = (toBeRemovedVersion |x| version)[pId] - (toBeInstalled |x| version)[pId]
toBeChanged = (toBeInstalled + toBeRemovedVersion)

objectives: minimize toBeRemovedPackage[count()], minimize toBeChanged[count()] // the paranoid criteria
```

Listing 2. Optimal Package Resolution in AlleAlle

Table 4. Relations for the optimal package resolution problem

Relation	Signature	Bound
installRequest	pId:id, relop:int, version:int	Exact
removeRequest	pId:id, relop:int, version:int	Exact
upgradeRequest	pId:id, relop:int, version:int	Exact
version	vId:id, pId:id, nr: int	Exact
installed	vId:id	Exact
keep	kId:id, vId:id	Exact
depends	vId:id, dcId:id	Exact
dependChoice	dcId:id, pId:id, version:int, relop:int	Exact
conflicts	vId:id, pId:id, version:int, relop:int	Exact
toBeInstalled	vId:id	Upper
toBeRemovedVersion	vId:id	Upper
toBeChanged	vId:id	Upper
toBeRemovedPackage	pId:id	Upper

of these relations are exactly those that are defined in the specification. The other 4 relations have an upper bound and represent the solution space for the solver to satisfy the user's request, given dependency and conflict constraints between package versions. The rest of the specification is shown in Listing 2, consisting of a mere 30 source lines of AlleAlle code.

In total we translated 57 of the competition problems, and measured the time spent in translating AlleAlle and running Z3 to solve the constraints. All found solutions by AlleAlle were correct and optimal, showing that the constructed specification is a correct implementation of the optimal dependency resolution problem according to the paranoid criteria. The full results can be found in Table 5, in Appendix B.

The results show that Z3 can efficiently solve the formula produced by AlleAlle, in the same order of magnitude as the winning solving times from the 2012 MISC competition. On the other hand, the time spent by AlleAlle translating the specification to SMT formula is high, ranging from 47 seconds to 62 minutes. Based on the specific problems exhibiting this behavior, we hypothesize that translation time correlates with the number of dependencies between the packages. The more dependencies between packages, the longer it takes AlleAlle to translate the problem to SMT formulas.

5.3 Comparing AlleAlle to Similar Systems

AlleAlle is a constraint solving system and language, comparable to Kodkod, Alloy, and SMT solvers. AlleAlle is an *intermediate* language: it is higher-level than the first-order logic formulas of SMT solvers like Z3 [22] or CVC4 [5] which are more general purpose logic solvers, but lower-level than, e.g., Alloy, which is a end-user, modeling language.

Kodkod is the back-end framework of Alloy, which is at the same level of abstraction as AlleAlle. Both AlleAlle and Kodkod support relational constraints, yet AlleAlle employs Codd's relational algebra making it possible to constraint data attributes directly (using the **where** operator), whereas Kodkod is based on Tarski's relation logic allowing constraints only to be expressed on the level of relations.

Although in terms of abstraction level, AlleAlle is comparable to Kodkod, the latter only exists as a Java library and does not feature a concrete syntax. AlleAlle's syntax allows us to experiment with specifications in a more flexible way, and thus may function as code generation target for higher-level languages, – essentially fulfilling the same role that Kodkod fulfills for Alloy.

AlleAlle leverages built-in theories of underlying SMT solvers, including support for optimization criteria available in solvers like Z3 [7]. Kodkod (and hence Alloy) require bit-encoding of integers because of their underlying SAT solvers, which is an impediment to performance for constraint problems that require such constraints. Optimization criteria are not available in either Alloy or Kodkod. For instance, the optimal dependency resolution problem (Section 5.2) cannot be expressed in either Alloy or Kodkod.

AlleAlle thus occupies a sweet spot in terms of both expressiveness (Codd's algebra) and solving performance (because of native SMT theories) between high-level languages like Alloy, and low-level relational solvers like Kodkod.

6 Related Work

SEM- and MACE-style model finders There are several finite model finding tools for first order logic (FOL). They can roughly be divided into two different groups: tools that implement specialized search strategies to find satisfying models (also known as SEM-style model finders) [27, 34] and tools that translate FOL formulas to SAT or SMT formulas and use an off-the-shelf solver to find satisfying instances (also known as MACE-style model finder) [9, 18, 25, 31]. AlleAlle falls in the MACE-style category.

Although all of these model finders accept FOL formulas as input not many of them accept relational logic. SEM [34] and FINDER [27] for instance accept a many-sorted logic of uninterpreted functions but no quantifiers. MACE2 [18], PARADOX [9], Fortress [33] and Razor [25] do allow for quantifiers but do not offer direct support for relational expressions.

Razor opts for a slightly different angle where it focuses on model exploration by searching for a minimal model first and allowing for model exploration using a predefined preorder relation. To achieve this Razor also exploits the SMT solver Z3 [25].

Fortress [33] also exploits an SMT solver by mapping FOL formulas to the logic of equality with uninterpreted functions (EUF). To make the model finite Vakilie et al. introduce so-called *range functions* to force restriction on the number of elements assigned to the different sorts [33].

Kodkod [31], the model finding engine used by Alloy [15, 30], accepts relational logic and transitive closure but offers no support for optimization criteria or first-class reasoning over data. As shown in the evaluation (see Section 5) Kodkod and Alloy do support integers but require the user to specify a fixed bit-width. Kodkod uses an explicit integer encoding which results in the introduction of more variables and ultimately into more CNF clauses in the generated SAT formula.

The other main difference with Kodkod is the interpretation of relational logic: Kodkod uses Tarski's definition of relational logic, AlleAlle uses Codd's relational algebra. The switch from Tarski to Codd prevents the need for an explicit encoding of data variables (i.e. integers) in the problem domain. E.g. in AlleAlle it is not needed to explicitly define the set of integers to be used in the relational expressions, it can directly encode constraints on the data attributes (using the `where` operator).

Relational reasoning with SMT El Ghazi et al. translate Alloy specifications to unbounded SMT constraints using an axiomatization of Alloy's relational logic in SMT, a so called shallow embedding of a relational theory [13]. Although this allows for proving some Alloy specifications it also struggles with many specifications, as is shown in later work [19].

Meng et al. define a relational calculus based on the theory of finite sets with cardinality constraints [4, 19]. This calculus is created explicitly to be implemented as a new theory in SMT solvers, a so called deep embedding, and contains many of the relational expressions that are also part of Kodkod (like join, transpose, product and transitive closure). The calculus does not require that bounds are set on the relations but uses earlier work by the authors on finite model finding [24]. The authors implement the calculus as a new theory in the SMT solver CVC4 [5] and evaluate its performance on existing benchmarks. Contrary to AlleAlle, the new theory is able to prove when a relational problem is unsatisfiable. Like Alloy and Kodkod, AlleAlle requires bounds on the relations with the result that any reported *unsat* only means that it is unsatisfiable with respect to the given bounds. The work by Meng et al. is based on the same calculus as Kodkod (Tarski's relational logic) meaning that reasoning on values of other theories (like integers) is limited in the same way as it is limited in Kodkod. Next to that, this work is implemented in CVC4 which currently does not support optimization objectives.

7 Conclusion

Relational model finding is a powerful technique that can be applied to a wide range of problems. In this paper we have introduced AlleAlle, a new language for describing such problems with first-class support for data attributes. AlleAlle specifications combine first-order logic with Codd's relational algebra, transitive closure, and optimization objectives. We have presented the formal semantics of AlleAlle and a detailed exposition of a novel algorithm to translate AlleAlle specifications to SMT constraints, to be solved by standard SMT solvers such as Z3.

Initial evaluation of our prototype implementation has shown that the performance leaves room for improvement. Both the translation speed and the generated formula efficiency can be improved. Theoretically it should be possible to improve the translation and solving time of pure relational problems to the level of Kodkod's performance since the translation algorithm of both model finders is comparable in terms of complexity. Early performance experiments

with a pure Java version of AlleAlle indeed hint into this direction.

AlleAlle supports compact encodings of relational problems with data. One example is optimal dependency resolution, as used in package managers. We have used this problem to assess expressiveness of AlleAlle. Although currently the translation times are high for these problems, the resulting SMT formulas can be solved efficiently by off-the-shelf solvers. The specification itself is concise, taking only half a page, which shows the expressiveness of combining relational logic, data, and optimization criteria.

There are multiple directions for future work. The current prototype of AlleAlle only supports integer domains; we will extend AlleAlle with support for reals, strings, and bit vectors since most SMT solves have built-in theories for these types. Another direction for further work is to include symmetry breaking predicates [9] to the generated SMT formulas, since this is well-known to have a positive effect on solver performance. Torlak et al. introduce a method to create these symmetry breaking predicates for relational logic [31], but it is an open question how this should be done in the presence of AlleAlle's data attributes. A final direction for further research is to investigate how *unsatisfiability core extraction* [29] can be used to explain reasons for `unsat` results. A particular challenge is how to map such explanations back to the level of AlleAlle.

A Algorithms

addDistinct The function addDistinct adds a tuple to a relation, ensuring that it will be distinct from other tuples in the relation by adding constraints to the `attCons` column:

1: **function** addDistinct(r: `rel`, t: `tuple`)
2: **for** $t' \leftarrow r_{body}$, $t \neq t'$, canOverlap(t,t') **do**
3: $t_{attCons} \leftarrow t_{attCons} \wedge (\neg t'_{exists} \vee \neg$attEqual$(t, t'))$
4: **end for**
5: $r_{body} \leftarrow r_{body} \cup \{t\}$
6: **return** r
7: **end function**

canOverlap The function canOverlap returns $\top$ when two tuples are indistinguishable with respect to their attribute values:

1: **function** canOverlap(t: `tuple`, t': `tuple`)
2: **for** $a \leftarrow t_{attributes}, a' \leftarrow t'_{attributes}, a_{name} = a'_{name}$ **do**
3: **if** $a_{value} \neq ? \wedge a'_{value} \neq ? \wedge a_{value} \neq a'_{value}$ **then**
4: **return** $\bot$ ▷ non-'holes', different values, overlap impossible
5: **end if**
6: **end for**
7: **return** $\top$
8: **end function**

attEqual The function attEqual constructs a constraint to ensure that two tuples will be equal.

1: **function** attEqual(t: `tuple`, t': `tuple`)
2: $\delta \leftarrow \top$
3: **for** $a \leftarrow t_{attributes}, a' \leftarrow t'_{attributes}, a_{name} = a'_{name}$ **do**
4: $\delta \leftarrow \delta \wedge a_{value} = a'_{value}$ ▷ force atts to have same values
5: **end for**
6: **return** δ
7: **end function**

B Optimal Package Dependency Resolution

Table 5. "MISC paranoid" AlleAlle results. Problem names refer to the problems as they were named in the original competition. See http://www.mancoosi.org/misc-2012/results/paranoid/ for an overview.

Problem name	Request type	# of Packages in CUDF	# dependencies	AlleAlle translation time (in sec)	Z3 solving time (in sec)	Best 2012 competition solving time (in sec)	Correct?	Optimal?
adf7b774-9af8-11df-bc37-00163e46d37a	upgrade	47203	22721	554.93	2.88	1.30	yes	yes
e0bd67a6-56d0-11df-b11f-00163e7a6f5e	upgrade	28333	14170	343.96	2.29	0.63	yes	yes
e2f6303a-4fe9-11e0-aa4f-00163e1e087d	install	41913	29726	1077.47	1.72	2.57	yes	yes
29180036-5408-11df-9f57-00163e7a6f5e	upgrade	28753	17490	403.31	1.22	0.99	yes	yes
7bf50d1c-9b1b-11df-8b50-00163e46d37a	upgrade	47210	17934	459.80	3.35	1.32	yes	yes
8b0e7c16-bab4-11e0-a883-00163e1e087d	install	59100	57811	2879.15	4.20	2.60	yes	yes
5698a62c-c731-11df-9bb9-00163e3d3b7c	install	54474	62705	3463.34	4.71	2.80	yes	yes
6b0d1da0-c730-11df-a7c5-00163e3d3b7c	install	54474	62792	3609.27	5.86	1.62	yes	yes
19890cfe-db9f-11df-9e6c-00163e3d3b7c	upgrade	33615	23979	743.49	3.24	1.15	yes	yes
978532fa-c730-11df-b070-00163e3d3b7c	install	54474	62792	3425.55	4.50	2.80	yes	yes
dd08e73e-d489-11df-b9cf-00163e3d3b7c	install	49561	48746	2230.60	3.20	1.78	yes	yes
d1583bd8-d489-11df-9a24-00163e3d3b7c	install	49561	48746	2327.53	3.68	1.44	yes	yes
dba3a3fe-3477-11e0-9e6c-00163e3d3b7c	upgrade	36310	17891	189.83	1.18	0.79	yes	yes
f4ebf9e0-360e-11e0-9e6c-00163e3d3b7c	upgrade	36213	18015	464.63	1.21	1.80	yes	yes
4ede8d96-c17a-11df-a7c5-00163e3d3b7c	install	51849	40000	1630.78	2.45	1.47	yes	yes
33bb2fbc-9512-11e0-9181-00163e1e087d	install	51134	15370	336.95	0.73	1.51	yes	yes
ab9005be-bacc-11e0-b0f6-00163e1e087d	install	59100	57811	2914.14	4.15	2.47	yes	yes
ff4a1d84-d490-11df-9e6c-00163e3d3b7c	install	53540	53307	2519.30	2.40	2.26	yes	yes
fa3d0fb2-db9e-11df-a0ec-00163e3d3b7c	upgrade	33615	23979	726.63	3.14	0.89	yes	yes
80e3fda2-9501-11e0-8001-00163e1e087d	install	51134	15370	342.71	0.74	1.57	yes	yes
d023d256-3477-11e0-bdb2-00163e3d3b7c	upgrade	36310	17891	445.69	2.57	1.10	yes	yes
103c9978-5408-11df-9bc1-00163e7a6f5e	upgrade	28753	17490	440.41	1.21	0.76	yes	yes
4a69cf16-c731-11df-9182-00163e3d3b7c	install	54474	62705	3761.11	4.74	1.62	yes	yes
26f3d4cc-d470-11df-9e6c-00163e3d3b7c	install	49561	48746	2272.93	3.28	1.65	yes	yes
ec32fc68-7254-11e0-8436-00163e1e087d	upgrade	39025	21410	575.76	1.33	1.50	yes	yes
cff22854-9512-11e0-8001-00163e1e087d	install	51134	15370	337.73	0.67	1.20	yes	yes
8680dd8a-8600-11e0-b285-00163e1e087d	install	53360	51425	2427.60	3.92	1.92	yes	yes
caefdef6-3477-11e0-84ef-00163e3d3b7c	upgrade	36310	17891	429.89	1.25	1.26	yes	yes
e381ba7e-a192-11e0-8647-00163e1e087d	install	51134	15370	349.90	0.79	1.54	yes	yes
deb285a6-db9e-11df-8f4f-00163e3d3b7c	upgrade	33615	23979	696.62	2.81	0.93	yes	yes
ca8f656c-db9e-11df-b9cf-00163e3d3b7c	upgrade	33615	23979	860.79	2.81	0.82	yes	yes
e599f3fc-360e-11e0-986e-00163e3d3b7c	upgrade	36213	18015	466.59	1.26	1.14	yes	yes
d0cc7514-c730-11df-a040-00163e3d3b7c	install	54474	62705	3392.81	4.50	2.90	yes	yes
bccf69ae-db9e-11df-9a24-00163e3d3b7c	upgrade	33615	23979	682.21	2.81	0.96	yes	yes
27000e82-c5c4-11df-a7c5-00163e3d3b7c	install	54485	62817	3393.98	4.14	1.34	yes	yes
d5026b8e-3477-11e0-986e-00163e3d3b7c	upgrade	36310	17891	448.12	1.20	0.94	yes	yes
e69a0e36-9ef1-11df-9d4a-00163e46d37a	install	50726	63860	3582.47	4.58	1.68	yes	yes
56e31304-c17a-11df-b070-00163e3d3b7c	install	51849	40000	1559.44	2.21	1.26	yes	yes
ed1cc19e-51b7-11e0-8436-00163e1e087d	install	42104	29996	948.50	1.32	1.66	yes	yes
a754ac72-95cc-11e0-9181-00163e1e087d	install	51134	15370	322.43	0.64	1.12	yes	yes
56ae4afa-0b33-11df-8a2b-00163e1d94dc	install	66940	4085	47.15	0.23	1.91	yes	yes
4e539b28-d46c-11df-8f4f-00163e3d3b7c	install	49561	48746	2309.19	3.20	1.50	yes	yes
fe523ea6-9b1b-11df-bc37-00163e46d37a	upgrade	47210	22707	573.45	3.40	0.95	yes	yes
b2540c52-51b7-11e0-aa4f-00163e1e087d	install	42104	29996	894.50	1.39	1.72	yes	yes
eeee44ce-5407-11df-b11f-00163e7a6f5e	upgrade	28753	17490	410.22	1.22	0.63	yes	yes
80cfe9a6-9b1b-11df-965e-00163e46d37a	upgrade	47210	17934	420.83	1.74	1.32	yes	yes
1aabfc32-d491-11df-9a24-00163e3d3b7c	install	53540	53307	2685.60	2.88	2.14	yes	yes
8222799a-9af8-11df-8b50-00163e46d37a	upgrade	47203	17933	422.87	1.70	1.22	yes	yes
301cbe92-a79c-11e0-9181-00163e1e087d	install	56865	55542	2798.53	0.00	0.00	yes	yes
7c834c0e-51b8-11e0-a49e-00163e1e087d	install	42104	29996	951.58	1.46	2.58	yes	yes
3e4f8550-0b33-11df-942d-00163e1d94dc	install	66940	4085	60.48	0.68	1.27	yes	yes
8afdd89e-51b8-11e0-acd7-00163e1e087d	install	42104	29996	897.15	1.31	1.66	yes	yes
4f84e9c6-a79c-11e0-9eb7-00163e1e087d	install	56865	55542	2785.50	0.00	0.00	yes	yes
7f80e4f0-4fe9-11e0-acd7-00163e1e087d	install	41913	29720	897.63	1.55	1.79	yes	yes
c2164c84-b015-11df-8b50-00163e46d37a	upgrade	32938	11545	236.60	0.33	1.11	yes	yes
0207e19a-9b1c-11df-af69-00163e46d37a	upgrade	47210	22707	571.12	2.67	1.30	yes	yes

Acknowledgments

We would like to thank our sheperd and reviewers for their constructive feedback which helped to improve this paper.

References

[1] P. Abate, R. Treinen, R. Di Cosmo, and S. Zacchiroli. MPM : a modular package manager. In *CBSE*, pages 179—-188. ACM, 2011.

[2] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *POPL*, pages 110–119. ACM, 1979.

[3] B. Bajić-Bizumić, C. Petitpierre, H.C. Huynh, and A. Wegmann. A model-driven environment for service design, simulation and prototyping. In *ESS*, pages 200–214. Springer, 2013.

[4] K. Bansal, A. Reynolds, C. Barrett, and C. Tinelli. A new decision procedure for finite sets and cardinality constraints in smt. In *CAV*, pages 82–98. Springer, 2016.

[5] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, Jovanović. D., T. King, A. Reynolds, and C. Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011.

[6] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *SMT*, volume 13, page 14, 2010.

[7] N. Bjørner, A. Phan, and L. Fleckenstein. *v*Z - An Optimizing SMT Solver. In *TACAS*, volume 15, pages 194–199, 2015.

[8] J. Brunel, D. Chemouil, L. Rioux, M. Bakkali, and F. Vallée. A viewpoint-based approach for formal safety & security assessment of system architectures. In *MoDeVVa*, volume 1235, pages 39–48, 2014.

[9] K. Claessen and N. Sörensson. New techniques that improve mace-style finite model finding. In *CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27. Citeseer, 2003.

[10] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[11] R. Di Cosmo. EDOS deliverable WP2-D2 . 1 : Report on Formal Management of Software Dependencies. Technical report, 2005. URL: https://hal.inria.fr/hal-00697463/document.

[12] C.J. Date. *An Introduction to Database Systems*. Reading, MA, Addison-Wesley, 6th edition, 1994.

[13] A.A. El Ghazi and M. Taghdiri. Relational reasoning via smt solving. In *FM*, pages 133–148. Springer, 2011.

[14] D. Grunwald, C. Gladisch, T. Liu, M. Taghdiri, and S. Tyszberowicz. Generating jml specifications from alloy expressions. In *HVC*, pages 99–115. Springer, 2014.

[15] D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT press, revised edition, 2012.

[16] S.A. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid. Testera: A tool for testing java programs using Alloy specifications. In *ASE*, pages 608–611. IEEE Computer Society, 2011.

[17] P. Klint, T. van der Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *SCAM*, pages 168–177. IEEE, 2009.

[18] W. McCune. A davis-putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Technical report, Argonne National Laboratory, 1994.

[19] B. Meng, A. Reynolds, C. Tinelli, and Clark Barrett. Relational constraint solving in smt. In *CAV*, pages 148–165. Springer, 2017.

[20] A. Milicevic and D. Jackson. Preventing arithmetic overflows in Alloy. *Science of Computer Programming*, 94:203–216, 2014.

[21] A. Milicevic, J.P. Near, E. Kang, and D. Jackson. Alloy*: a general-purpose higher-order relational constraint solver. In *ICSE*, pages 609–619. IEEE Press, 2015.

[22] L. De Moura and N. Bjorner. Z3: An efficient SMT solver. *TACAS*, pages 337–340, 2008.

[23] C. Prud'homme, J.G. Fages, and X. Lorca. *Choco Documentation*, 2017. URL: http://www.choco-solver.org.

[24] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in smt. In *CAV*, pages 640–655. Springer, 2013.

[25] S. Saghafi. *A Framework for Exploring Finite Models*. PhD thesis, Worcester Polytechnic Institute, 2015.

[26] R. Sebastiani and P. Trentin. Optimathsat: a tool for optimization modulo theories. In *CAV*, pages 447–454. Springer, 2015.

[27] J. Slaney. Finder: Finite domain enumerator system description. In *CAV*, pages 798–801. Springer, 1994.

[28] E. Torlak. *A Constraint Solver for Software Engineering : Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, 2009.

[29] E. Torlak, F. Chang, and D. Jackson. Finding minimal unsatisfiable cores of declarative specifications. In *FM*, pages 326–341. Springer, 2008.

[30] E. Torlak and G. Dennis. Kodkod for Alloy users. In *Alloy Workshop*, 2006.

[31] E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, pages 632–647. Springer, 2007.

[32] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.

[33] A. Vakili and N.A. Day. Finite model finding using the logic of equality with uninterpreted functions. In *FM*, pages 677–693. Springer, 2016.

[34] J. Zhang and H. Zhang. Sem: a system for enumerating models. In *IJCAI*, volume 95, pages 298–303, 1995.