Quantitative Relational Modelling with QAlloy

Pedro Silva
INESC TEC & Universidade do Minho
Braga, Portugal
pedro.d.silva@inesctec.pt

José N. Oliveira
INESC TEC & Universidade do Minho
Braga, Portugal
jno@di.uminho.pt

Nuno Macedo
INESC TEC & Universidade do Porto
Porto, Portugal
nmacedo@fe.up.pt

Alcino Cunha
INESC TEC & Universidade do Minho
Braga, Portugal
alcino@di.uminho.pt

ABSTRACT

Alloy is a popular language and tool for formal software design. A key factor to this popularity is its *relational logic*, an elegant specification language with a minimal syntax and semantics. However, many software problems nowadays involve both structural and quantitative requirements, and Alloy's relational logic is not well suited to reason about the latter. This paper introduces QAlloy, an extension of Alloy with *quantitative relations* that add integer quantities to associations between domain elements. Having integers internalised in relations, instead of being explicit domain elements like in standard Alloy, allows quantitative requirements to be specified in QAlloy with a similar elegance to structural requirements, with the side-effect of providing basic dimensional analysis support via the type system. The QAlloy Analyzer also implements an SMT-based engine that enables quantities to be unbounded, thus avoiding many problems that may arise with the current bounded integer semantics of Alloy.

CCS CONCEPTS

• **Software and its engineering** → **Specification languages**; **Formal methods**; **Software system models**.

KEYWORDS

Alloy, quantitative modelling, SMT, model finding, linear algebra, relational specifications

ACM Reference Format:
Pedro Silva, José N. Oliveira, Nuno Macedo, and Alcino Cunha. 2022. Quantitative Relational Modelling with QAlloy. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549154

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549154

1 INTRODUCTION

Employing trustworthy software design techniques early in the development process is crucial to avoid critical faults in later stages. This process often focuses on structural and architectural modelling, but there is a trend towards *quantitative modelling* in the software sciences. In the words of the editors of [2]:

> *(…) Today there are many quantitative aspects of system design: they include timing (whether discrete, continuous or hybrid); probabilistic aspects of success or failure including cost and reward; and quantified information flow.*

Moreover, data science has emerged as a very important discipline in computing, which is quantitative by definition. While software modelling relies mainly on logics and relational algebra, data analysis relies mostly on linear algebra [1]. The need for a *linear algebra of programming* has been identified as a possible way to extend standard software design techniques to such quantitative fields [16]. This work goes in a similar direction.

Alloy [9] is a lightweight formal specification language based on relational logic, which is supported by an automatic Analyzer[1]. The flexibility and simplicity of the language, together with the quick and intuitive feedback provided by the Analyzer, have led to the wide adoption of Alloy in the formal validation and verification of software design models. Unfortunately, while Alloy's relational logic has shown to be well-suited to reason about structural problems, it is rather limited for reasoning about quantitative models.

Some of these issues arise at the language level. Although integers in Alloy are a special kind of atom, they mostly act as other uninterpreted atoms in the relational formalism. This has some unintuitive consequences due to the uniqueness of elements in sets, and undermines the ability of Alloy's rich type system [6] to catch specification errors. Other issues are due to the associated analysis procedures. The Analyzer supports either wrap-around semantics with a necessarily reduced precision due to the SAT-based backend, or non-standard semantics that prevent overflows [13] which has unpredictable side-effects. While work has been developed to tackle some of these issues, such as supporting multi-relations [19] or allowing an unbounded integer domain through SMT-based backends [7, 12, 18], there is no unified and principled approach to quantitative modelling and analysis based on Alloy.

[1] This work is based on version 5 of Alloy and still does not consider the temporal features recently introduced in Alloy 6.

```
1  sig Bag {
2    contains : Int one → Product
3  }
4  abstract sig Product {
5    stock  : one Int,
6    weight : one Int
7  }
8  one sig Tea, Coffee, Milk extends Product {}
9  fact {
10   all b : Bag, p : Product | b.contains.p ≥ 0
11   all p : Product | p.weight ≥ 1
12   all p : Product | p.stock ≥ 0 and p.stock ≤ 3
13   Milk.weight = 10
14   Milk.weight > Coffee.weight
15   Coffee.weight = mul[3,Tea.weight]
16   all p : Product |
17     p.stock ≥ (sum b : Bag | b.contains.p)
18 }
```

Figure 1: SCO model in Alloy

Yet, Alloy offers a particularly interesting setting for such a quantitative extension, amounting to extending its underlying Boolean matrices to numeric ones, hiding the numeric calculations "under the carpet" while keeping its notational elegance at a high-level. Following on this vision, this paper presents QAlloy and its Analyzer. QAlloy is a minimal extension to the Alloy language that internalizes quantities in the relations without sacrificing the simplicity, flexibility and high-level of abstraction of the Alloy language. QAlloy allows relations to be declared as quantitative, and the relational operators have been generalized to this setting. Integers are no longer elements of the universe of discourse, but rather "measures" of relationships. In practice, this means that quantities can be associated with units of measure, allowing the type checker to detect dimensional inconsistencies and forcing a disciplined use of integers when modelling. Internally, this entailed moving from a formalization based on Boolean matrices to one based on integer matrices, and adapting the backend to be SMT-based rather than SAT-based, naturally allowing unbounded quantities.

The rest of the paper is structured as follows. Section 2 presents QAlloy through a motivating example. Section 3 presents its syntax and semantics, while Section 4 presents the SMT-based analysis backend. Section 5 evaluates the flexibility and performance of QAlloy. Section 6 discusses relevant related work, and Section 7 wraps up the paper with conclusions and directions for future work.

2 MOTIVATING EXAMPLE

Alloy excels at describing and exploring structures. One typical application is domain modelling, where the goal is to describe entities and their relationships, and elicit the requirements that govern them. Take for example a simple (partial) domain model of a supermarket self-checkout (SCO) system. Some of the relevant entities in this domain are shopping *bags* and the different *products* on sale. Several quantities, of different units, are also relevant: the quantity of products each bag *contains*, the current *stock* of products (to issue alerts for stock shortages), and the *weight* of each product (to confirm that a given item was placed inside the bag). The first and second are measured in number of items, while the third is measured in ounces.

2.1 Quantitative Modelling with Alloy

The SCO can be encoded in Alloy as presented in Figure 1. Entities can be modelled in Alloy by declaring *signatures*, which are sets of elements drawn from the universe of discourse. An optional multiplicity can be used before the **`sig`** keyword to restrict the cardinality of the declared signature. Signatures can also be structured in a hierarchy, with disjoint sub-signatures being declared with keyword **`extends`**. The parent signature can be declared as **`abstract`** if it should not contain elements besides those in its extensions. This is the case of signature `Product` which has three singleton extension sub-signatures, each representing a different product on sale.

The most natural way to model quantities in Alloy is to specify them explicitly using integers. Alloy has a pre-defined **`Int`** signature that contains all the integers that can be represented with a given bit-width using two's complement representation.

Inside a signature it is possible to declare *fields*, relations (i.e., sets of tuples) that connect elements of the parent signature to elements of other signatures. For each of the above quantities there is a field in the model that relates the relevant signatures to exactly one **`Int`**: fields `weight` and `stock` are binary relations that associate each `Product` with the respective weight and stock, and field `contains` is a ternary relation that associates each `Bag` and `Product` to the number of items of that product the bag contains.

Constraints are specified using *relational logic*, an extension of first-order logic with relational operators. The most used relational operator is *dot join* (**`.`**) that *composes* two relations. To simplify the syntax and semantics, in Alloy everything is a relation. In particular, signatures are unary relations – sets of tuples with a single element – and scalars and quantified variables are singletons. This means that operators like dot join can be used not only to compose two fields, but also variables and signatures with fields. For example, if `p` is a product and `b` a bag, `b.contains.p` is the number of items of product `p` inside bag `b`.

In our model, a **`fact`** contains the various assumptions in our domain. The first three constraints (lines 10–12) force quantities to be non-negative, weights to be positive, and impose an upper limit on the current stock of products. The next three constraints (lines 13–15) impose (rather) loose restrictions on the weight of the three different products. The final constraint (lines 16–17) restricts the total quantity of a product in all bags to be less or equal than its current stock. One of the drawbacks of using integers in Alloy is that simple constraints such as these are not trivial to specify. The problem is that, since all expressions denote sets, repeated quantities are not properly accounted when using composition. For example, expression `Bag.contains.p` collects the quantities of `p` in all bags, but if two bags have the same quantity of `p` that integer will appear only once in the final set. This means that to compute this value we need to use the special **`sum`** quantifier, that sums all expressions ranging over a set.

Still related to language support, a more fundamental drawback is lack of typing for units. Alloy's type system is quite good at catching specification errors [6]. For example, expression `contains.Bag` would raise a warning since bags cannot contain bags. By using **`Int`** to represent all kinds of quantities, not taking into account the

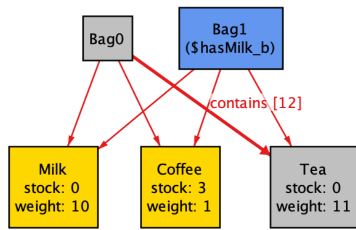

Figure 2: Bogus Alloy counter-example to `hasMilk`

respective units, the type system does not help in detecting silly constraints such as `all p : Product | p.stock ≥ p.weight`.

Alloy models can include **run** commands to ask for an *instance* of the model (a valid assignment to all declared signatures and fields) or **check** commands to verify assertions (which returns a counter-example instance if the assertion is invalid). For decidability, the analysis performed by these commands is *bounded*: the universe of discourse is finite and its size can be controlled by a *scope* imposed on signatures, that defines the maximum number of elements they can contain. An exception is the scope on **Int** that defines the bit-width of the integers created in the universe of discourse. This means that a **check** command may fail to find a counter-example to an invalid assertion. But, since most invalid assertions can be refuted with small counter-examples – the so called *small scope hypothesis* – the bounded analysis implemented by the Alloy Analyzer is still useful in practice to achieve a high-level of confidence.

Moreover, the Analyzer allows the user to iterate over the instances returned by these commands (to see alternative scenarios or counter-examples), and also depicts them as graphs for easier comprehension[2]. These features make **run** commands extremely useful, since they allow the user to easily explore design alternatives and discover missing requirements.

Consider, for example the following assertion, that checks if bags weighting more that 30 oz necessarily contain milk cartons inside.

```
assert hasMilk { all b : Bag |
    (sum p : Product | mul[b.contains.p, p.weight]) > 30
    implies b.contains.Milk ≥ 1 }
```

This assertion is invalid, the "lightest" counter-example being a bag with three 9 oz coffee bags and two 3 oz tea packets, with total weight 33 oz. To verify it we could have the following command that sets a scope of 2 to all signatures and a bit-width 5 for **Int**[3].

```
check hasMilk for 2 but 5 Int
```

By executing this command we get the bogus counter-example in Figure 2, where one bag contains 12 milk cartons, although there are none in stock. The weight of coffee bags is also not three times the weight of tea packets, as specified in the fact.

This points to another drawback of Alloy when dealing with quantities – the default semantics for integer operations is *wrap around*, hence the above bogus counter-example (the other bag in the counter-example had 11 milk cartons, which added to 12 overflows with bit-width 5 and yields a negative number). The Analyzer currently also implements a semantics that *prevents overflows* for integers [13], which would need to be activated to verify this assertion. However, rerunning the command now yields no counter-examples. This time the reason is rather pedantic: the specified bit-width is not big enough to represent the constant 30, so all instances overflow when evaluating the left-hand-side of the implication in the assertion and are discarded. This means that the user has to be very careful when setting the scope for **Int**, at the risk of easily missing possible counter-examples. In fact, even increasing the bit-width to 6, which already is enough to represent all integer constants in the model, would not suffice to falsify the assertion, since the "lightest" counter-example bag has total weight that exceeds 32 oz. In this case we would need a bit-width of at least 7. This choice is further complicated by the complexities of the *prevent overflows* semantics. One might think that the solution would be to set a rather large bit-width, but unfortunately that is not viable in many situations. On the one hand, it would slow down analysis considerably. On the other hand, with a bit-width larger than 10 the universe would be too large to allow the representation of ternary relations in the SAT-based analysis engine, rendering assertion verification impossible.

2.2 Quantitative Modelling with QAlloy

QAlloy improves the handling of integers in Alloy, addressing all the above drawbacks. First, instead of having an explicit **Int** signature, QAlloy internalises integers in relations – while Alloy relations are matrices of Booleans that determine which tuples belong to the relation, in QAlloy it is possible to declare relations that are matrices of integers, which we will denote by *quantitative relations*, where each tuple is paired with a quantity. QAlloy also generalizes integer operations to work on quantitative relations and provides a new composition operator that implements matrix multiplication, allowing easier specification of constraints involving quantities. Second, this internalisation of integers in quantitative relations allows them to inherit the type stated in the declaration. This means that the standard Alloy type system can now detect meaningless constraints where integers of different units of measure are compared. Finally, the QAlloy Analyzer uses SMT solvers instead of SAT solvers in the analysis backend in order to support unbounded integers. This means that the semantics of integer operations is now straightforward, and the user no longer needs to worry about determining the correct bit-width.

The SCO can be modelled in QAlloy as shown in Figure 3. Quantitative relations are declared with keyword **int**. For example, `contains` is now a binary quantitative relation that associates each bag with the quantity of each product it contains. To model the weight we first introduce a singleton signature `Oz` to model the respective unit, and then declare `weight` as a binary quantitative relation that relates each product to the quantity of ounces it weights. It is also possible to declare quantitative subset signatures (declared with keyword **in**, likewise in Alloy). That is the case of `stock`, a quantitative subset of `Product`. Quantitative subsets are vectors that pair each element of the parent signature with a quantity.

[2]These graphs can also be customised using themes. The instances show in this paper use custom themes to make them easier to understand.
[3]Note that the Analyzer automatically grows the scope of `Product` to 3 to accommodate the declared singleton extensions.

```
1  sig Bag {
2    int contains : set Product
3  }
4  one sig Oz {}
5  abstract sig Product {
6    int weight : one Oz
7  }
8  one sig Tea, Coffee, Milk extends Product {}
9  int sig stock in Product {}
10 fact {
11   contains ≥ 0 ** (Bag → Product)
12   weight ≥ 1 ** (Product → Oz)
13   stock ≥ 0 ** Product and stock ≤ 3 ** Product
14   Milk;weight = 10 ** Oz
15   Milk;weight > Coffee;weight
16   Coffee;weight = 3 ** (Tea;weight)
17   Bag;contains ≤ stock
18 }
```

Figure 3: SCO model in QAlloy

In QAlloy, integer operators, such as comparisons (≤ or ≥) and arithmetic operations (**add** or **mul**), have been lifted to work on quantitative relations, operating entry-wise in the respective matrices or vectors. For example, **add** implements matrix addition and **mul** implements the Hadamard product. In addition, the new *quantitative composition* operator (`;`) implements matrix multiplication. These operators can also be used with non-quantitative relations and signatures, treating them as binary quantitative matrices. Integer constants can no longer be used standalone, but only in the left-hand-side of the new *scalar multiplication* (`**`) operator. The reason for this is to achieve type-safety. For example, to specify a quantity of 10 oz, one should write `10 ** Oz`, while a quantity of 10 tea packets would be specified as `10 ** Tea`. A comparison such as `10 ** Oz = 10 ** Tea` yields a type error. To specify a binary quantitative relation from bags to products with a quantity of 10 in all possible tuples, one should first compute the Cartesian product `Bag → Product`, that attaches a quantity of 1 to all possible pairs, and than use scalar multiplication to scale up those quantities, as in `10 ** (Bag → Product)`. The normal Alloy operators, such as dot join, intersection (&), or union (+), also work with quantitative relations, implementing a min-max algebra inspired by multi-sets, which are just a special case of quantitative relations where all quantities are non-negative. If used only with non-quantitative relations, these have the same semantics as before, meaning that Alloy models that do not use integers are retro-compatible with QAlloy.

The first three constraints (lines 11–13) in Figure 3, specify that all the quantitative relations in this example are in fact multi-sets, with non-negative quantities. Like before, the weight is restricted to be strictly positive and the stock of each product is limited to 3 items. Note that, since comparisons operate entry-wise in quantitative relations we do not need quantifiers to specify these properties, and can adopt a more terse (*point-free*) style of specification. The next three constraints (lines 14–16) specify the weight restrictions of the different products. To determine the weight of a given product we should use the quantitative composition operator: while `Milk.weight` determines if milk weights something, `Milk;weight` determines its actual weight. Notice that `Milk` is a vector with a 1 in

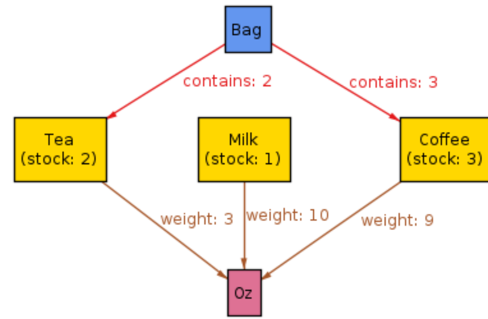

Figure 4: QAlloy counter-example to `hasMilk`

the entry corresponding to milk, while `weight` is a matrix that only has positive quantities in the `Oz` column and the `Product` rows. Hence, their multiplication yields a vector that has a single positive quantity in the `Oz` entry. The last constraint (line 17) shows the advantage of the new quantitative composition, enabling a very elegant specification of the requirement that the total products in all bags does not exceed the current stock.

Lastly, the `hasMilk` assertion can be specified as follows.

```
assert hasMilk { all b : Bag |
  b;contains;weight > 30 ** Oz
  implies b;contains ≥ 1 ** Milk }
```

To verify this assertion we could use a **check** `hasMilk` **for** 2: the SMT backend analysis that implements unbounded integer semantics would immediately return the counter-example in Figure 4. If we increase the weight to `40 ** Oz` the assertion becomes valid (due to the current limits on the stock) and the QAlloy Analyzer no longer returns any counter-example.

3 SYNTAX AND SEMANTICS

The proposed language is a minor adaptation of the Alloy language. Its concrete syntax is presented in Figure 5, with additions highlighted (underlined) and references to standalone integer constants removed.

As shown in the previous section, structure is introduced through the declaration of non-quantitative (or *qualitative*) *signatures* (keyword **sig**), which represent sets of uninterpreted atoms. A hierarchy can be introduced through *extension* (keyword **extends**), which imposes sub-signatures to be disjoint; if a signature is declared as **abstract**, all atoms must belong to some sub-signature.

Relations of arbitrary arity (including sets – unary relations) can be defined over these signatures. Sets are introduced by declaring *inclusion* signatures (keyword **in**) and relations through the declaration of fields within signatures. Such relations can be declared as *quantitative* with the keyword **int**. Over all such declarations, multiplicity constraints can be enforced (keywords **some**, **lone** and **one**), controlling the number of tuples they may contain. In qualitative relations these multiplicity constraints affect the cardinality of tuple sets. In quantitative relations, a tuple is present if it has a nonzero quantity, so multiplicity constraints restrict the number of tuples with nonzero quantity. For instance, a subset declared as **one int** will have exactly one element with an arbitrary nonzero

```
spec ::= module qualName [ [ name ,+ ] ] import* paragraph*
import ::= open qualName [ [ qualName ,+ ] ] [ as name ]
paragraph ::= sigDecl | factDecl | funDecl | predDecl
    | assertDecl | checkCmd
sigDecl ::= [ int ] [ abstract ] [ mult ] sig name ,+
                  [ sigExt ] { intDecl ,* } [ block ]
sigExt ::= extends qualName | in qualName [ + qualName ]*
mult ::= lone | some | one
decl ::= [ disj ] name ,+ : [ disj ] expr
intDecl ::= [ int ] decl
factDecl ::= fact [ name ] block
assertDecl ::= assert [ name ] block
funDecl ::= fun name [ [ decl ,* ] ] : expr { expr }
predDecl ::= pred name [ [ decl ,* ] ] block
expr ::= const | qualName | @name | this | unOp expr
    | expr binOp expr | expr arrowOp expr
    | expr [ expr ,* ] | expr [ ! | not ] compareOp expr
    | expr ( => | implies ) expr else expr
    | quant decl ,+ blockOrBar | ( expr ) | block
    | { decl ,+ blockOrBar }
const ::= none | univ | iden
unOp ::= ! | not | no | mult | set | ~ | * | ^ | #
    | drop | number **
binOp ::= || | or | && | and | <=> | iff | => | implies
    | & | + | − | ++ | <: | :> | . | ;
    | add | sub | mul | div | rem
arrowOp ::= [ mult | set ] → [ mult | set ]
compareOp ::= in | = | != | < | ≤ | > | ≥
letDecl ::= name = expr
block ::= { expr* }
blockOrBar ::= block | | expr
quant ::= all | no | mult
checkCmd ::= check qualName [ scope ]
scope ::= for number [ but typescope ,+ ] | for typescope ,+
typescope ::= [ exactly ] number qualName
qualName ::= [ this/ ] ( name/ )* name
```

Figure 5: Concrete syntax of the QAlloy language.

quantity. Some constant relations are also available, namely **none** (empty set), **univ** (non-quantitative set of all atoms), and **iden** (identity relation).

Relational expressions are built using typical relational operators adapted to the quantitative context. The formal semantics for a kernel of QAlloy expressions is given by the *quantity function*[4] defined inductively over relational expressions in Figure 6. Let $\mathcal{A}$ be the universe of atoms. A *binding* is a function that for every free relation (signatures and fields) r and tuple t with appropriate arity returns its quantity, i.e., if $s(r, t) = q$, t has quantity q in r. Then, the quantity function of a relational expression Γ under binding s is given by $[\![\Gamma]\!]_s$. We say that a tuple t belongs to Γ (under binding s) if $[\![\Gamma]\!]_s(t) \neq 0$. The universe of atoms $\mathcal{A}$ is calculated from the scopes defined in the command under analysis. Based on the command scope and on their type, each declared signature and field is assigned an *upper-bound* on $\mathcal{A}$, the set of tuples that may belong to a relation. The upper-bound of any relational expression can then be derived by applying the operators at the upper-bound level. For instance, if relations `r` and `s` have upper-bounds `{(A0),(A1)}` and `{(B1),(B2)}`, respectively, then the union `r+s` has upper-bound `{(A0),(A1),(B0),(B1)}`. We denote the upper-bound of a relational expression Γ by $\lceil\Gamma\rceil$.

[4]We abstain from using the more common term "multiplicity function" to avoid confusion with Alloy's multiplicity constraints over unique atom tuples.

QAlloy provides two different classes of operations over quantitative relations. The first interprets quantitative relations as multirelations that also allow negative quantities (following [4]), which degenerates back into regular relational operations when quantities are restricted to 0 or 1. In this class we have the *union* (+), *intersection* (&), and *difference* (−) operations, that represent, respectively, the entry-wise nonzero maximum, minimum, and subtraction. We denote the nonzero maximum and minimum as $\max^0$ and $\min^0$, respectively, and define them as:

$$\begin{aligned} \min^0(x,0) &= x & \max^0(x,0) &= x \\ \min^0(0,y) &= y & \max^0(0,y) &= y \\ \min^0(x,y) &= \min(x,y) & \max^0(x,y) &= \max(x,y) \end{aligned}$$

In the second class we have the arithmetic Alloy operations **add**, **sub**, **mul**, **div**, and **rem**, which are extended to represent the *entry-wise addition*, *subtraction*, *multiplication*, and integer *division* and *remainder*. *Scalar multiplication* (`**`) between an integer and a relation is also available.

Two versions of the quintessential relational *composition* are also available. In general, a tuple is in the composition of two relations Γ and Δ if there is a middle element to which Γ has an outgoing transition and Δ an incoming transition. The *dot join* (`.`) generalizes the Boolean version, and the quantity of a transition from Γ to Δ through a middle common element is the nonzero minimum of the incoming and outgoing transitions; the quantity of a tuple in $\Gamma\,.\,\Delta$ is the nonzero maximum among all such possible transitions. The *quantitative join* (`;`) implements matrix multiplication, and the quantity of a transition through a middle element is the product of the quantities of the incoming and outgoing transitions; the quantity of a tuple in $\Gamma;\Delta$ is the sum of all such transitions. Note that **iden** is the neutral element of `;`, but for `.` it is only the neutral element when the other argument is non-quantitative.

The *Cartesian product* (→) of any two relations multiplies the quantities of the originating tuples. A binary relation can also be *reversed* (~), preserving the quantities of the tuples. The *transitive closure* (^) of a binary relation is defined through the iterative application of dot join, and the *reflexive transitive closure* (*) through the union of the transitive closure with **iden**. Derived operators, such as *override* (++) and *domain* (<:) and *range* (:>) restriction have the same definition as in Alloy. Relations can also be defined by *comprehension*. Any relation can also be dropped (**drop**) to its non-quantitative representation, where all nonzero quantities are mapped to 1.

Lastly, the *cardinality* operator (#) is the summation of all quantities in a relation Γ, with the resulting quantity being attached to all tuples in its upper-bound $\lceil\Gamma\rceil$, to obtain a constant that preserves units of measure. Notice that, other than in the scalar multiplication, integer constants are no longer valid in QAlloy. This is part of our vision towards the disciplined use of quantities. In particular, if the total quantity of $\#\Gamma$ is q, the resulting expression is the same as q`**`$\lceil\Gamma\rceil$.

Relations can then be combined into formulas. The formal semantics for a kernel of QAlloy formulas is presented in Figure 7. For a binding s, the semantics of a formula ϕ is given by $s \models \phi$.

$$
\begin{array}{lcl}
[\![r]\!]_s(t) & = & s(r,t) \\
[\![x]\!]_s((a)) & = & s(x,(a)) \\
[\![\mathbf{univ}]\!]_s((a)) & = & 1 \\
[\![\mathbf{none}]\!]_s((a)) & = & 0 \\
[\![\mathbf{iden}]\!]_s((a,b)) & = & \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases} \\
[\![\Gamma + \Delta]\!]_s(t) & = & \max^0([\![\Gamma]\!]_s(t), [\![\Delta]\!]_s(t)) \\
[\![\Gamma \,\&\, \Delta]\!]_s(t) & = & \min^0([\![\Gamma]\!]_s(t), [\![\Delta]\!]_s(t)) \\
[\![\Gamma - \Delta]\!]_s(t) & = & \begin{cases} 0 & \text{if } [\![\Gamma]\!]_s(t) = 0 \\ [\![\Gamma]\!]_s(t) - \min([\![\Gamma]\!]_s(t), [\![\Delta]\!]_s(t)) & \text{otherwise} \end{cases} \\
[\![\mathbf{add}[\Gamma,\Delta]]\!]_s(t) & = & [\![\Gamma]\!]_s(t) + [\![\Delta]\!]_s(t) \\
[\![\mathbf{sub}[\Gamma,\Delta]]\!]_s(t) & = & [\![\Gamma]\!]_s(t) - [\![\Delta]\!]_s(t) \\
[\![\mathbf{mul}[\Gamma,\Delta]]\!]_s(t) & = & [\![\Gamma]\!]_s(t) \times [\![\Delta]\!]_s(t) \\
[\![\mathbf{div}[\Gamma,\Delta]]\!]_s(t) & = & [\![\Gamma]\!]_s(t) \;/\; [\![\Delta]\!]_s(t) \\
[\![\mathbf{rem}[\Gamma,\Delta]]\!]_s(t) & = & [\![\Gamma]\!]_s(t) \;\%\; [\![\Delta]\!]_s(t) \\
[\![n \,{**}\, \Gamma]\!]_s(t) & = & n[\![\Gamma]\!]_s(t) \\
[\![\Gamma \,.\, \Delta]\!]_s((a_1,\ldots,a_{n-1},b_2,\ldots,b_m)) & = & \max^0_{c \in \mathcal{A}} (\min^0([\![\Gamma]\!]_s((a_1,\ldots,a_{n-1},c)), [\![\Delta]\!]_s((c,b_2,\ldots,b_m)))) \\
[\![\Gamma \,;\, \Delta]\!]_s((a_1,\ldots,a_{n-1},b_2,\ldots,b_m)) & = & \sum_{c \in \mathcal{A}} ([\![\Gamma]\!]_s((a_1,\ldots,a_{n-1},c)) \times [\![\Delta]\!]_s((c,b_2,\ldots,b_m))) \\
[\![\Gamma \rightarrow \Delta]\!]_s((a_1,\ldots,a_n,b_1,\ldots,b_m)) & = & [\![\Gamma]\!]_s((a_1,\ldots,a_n)) \times [\![\Delta]\!]_s((b_1,\ldots,b_m)) \\
[\![{\sim}\Gamma]\!]_s((a,b)) & = & [\![\Gamma]\!]_s((b,a)) \\
[\![{\wedge}\Gamma]\!]_s((a,b)) & = & \max^0([\![\Gamma]\!]_s((a,b)), [\![\Gamma \,.\, \Gamma]\!]_s((a,b)), [\![\Gamma \,.\, \Gamma \,.\, \Gamma]\!]_s((a,b)), \ldots) \\
[\![\{ x_1 : \Gamma_1, \ldots, x_n : \Gamma_n \mid \phi \}]\!]_s((a_1,\ldots,a_n)) & = & \begin{cases} [\![\Gamma_1]\!]_s((a_1)) \times \ldots \times [\![\Gamma_n]\!]_s((a_n)) & \text{if } s' \models \phi \\ 0 & \text{otherwise} \end{cases} \\
 & & \quad \text{where } s' = s \oplus (x_1,(a_1)) \mapsto [\![\Gamma_1]\!]_s((a_1)) \oplus \ldots \oplus (x_n,(a_n)) \mapsto [\![\Gamma_n]\!]_s((a_n)) \\
[\![\mathbf{drop}\ \Gamma]\!]_s(t) & = & \begin{cases} 0 & \text{if } [\![\Gamma]\!]_s(t) = 0 \\ 1 & \text{otherwise} \end{cases} \\
[\![\#\Gamma]\!]_s(t) & = & \begin{cases} \sum_{t' \in \lceil \Gamma \rceil} [\![\Gamma]\!]_s(t') & \text{if } t \in \lceil \Gamma \rceil \\ 0 & \text{otherwise} \end{cases}
\end{array}
$$

Figure 6: Semantics of QAlloy relational expressions ($\mathcal{A}$ is the declared universe, n and m the arity of Γ and Δ, respectively)

$$
\begin{array}{lcl}
s \models \Gamma\ \mathbf{in}\ \Delta & \equiv & \forall t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) \neq 0 \Rightarrow [\![\Delta]\!]_s(t) \neq 0 \wedge [\![\Gamma]\!]_s(t) \leq [\![\Delta]\!]_s(t) \\
s \models \Gamma \leq \Delta & \equiv & \forall t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) \leq [\![\Delta]\!]_s(t) \\
s \models \Gamma \geq \Delta & \equiv & \forall t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) \geq [\![\Delta]\!]_s(t) \\
s \models \Gamma < \Delta & \equiv & \forall t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) \leq [\![\Delta]\!]_s(t) \wedge \exists t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) < [\![\Delta]\!]_s(t) \\
s \models \Gamma > \Delta & \equiv & \forall t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) \geq [\![\Delta]\!]_s(t) \wedge \exists t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) > [\![\Delta]\!]_s(t) \\
s \models \mathbf{some}\ \Gamma & \equiv & \exists t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) \neq 0 \\
s \models \mathbf{lone}\ \Gamma & \equiv & \forall t_1, t_2 \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t_1) \neq 0 \wedge [\![\Gamma]\!]_s(t_2) \neq 0 \Rightarrow t_1 = t_2 \\
s \models \mathbf{not}\ \phi & \equiv & s \not\models \phi \\
s \models \phi\ \mathbf{and}\ \psi & \equiv & s \models \phi \wedge s \models \psi \\
s \models \mathbf{all}\ x : \Gamma \mid \phi & \equiv & \forall t \in \lceil \Gamma \rceil \cdot [\![\Gamma]\!]_s(t) \neq 0 \Rightarrow s \oplus (x,t) \mapsto [\![\Gamma]\!]_s(t) \models \phi
\end{array}
$$

Figure 7: Semantics of QAlloy formulas

The fundamental atomic formula is the inclusion test between two relations (**in**). In multi-relations with negative quantities [4], Γ **in** Δ tests whether every tuple in Γ exists in Δ and with a smaller, or equal, quantity. The arithmetic comparison operators $\leq$ and $\geq$ are extended to the entry-wise comparison, while $<$ and $>$ requires that at least one entry is strictly less/greater. The other atomic formulas are multiplicity constraints over the number of atoms in a relation (again, regardless of their quantity). Formulas are then combined

with regular Boolean operators **not**/!, **and**/&&, **or**/||, **implies**/=> or **iff**/<=>, or universal (**all**) and existential (**some**) first-order quantifications. In the latter, quantified variables are assigned their quantity in the quantification domain.

One characteristic of this semantics is that, when restricted to qualitative relations, it degenerates into that of standard Alloy.

Theorem 3.1 (Alloy/QAlloy equivalence). *A model without quantitative aspects (one that is included in the intersection of both languages – i.e., without quantitative relations and operators, references to* **Int***, standalone integer constants, and integer operators) has the same semantics in Alloy and QAlloy.*

Proof. Without quantitative relations all quantities in tuples will be restricted to 0 or 1 and all non-quantitative operators preserve such binary quantities, having the same semantics as in Alloy (with 1 denoting membership of the respective tuple to the relation). Operators +, &, – and **in** over non-quantitative relations correspond to their regular set version [4]. The inner $\min^0$ in `.` returns 1 whenever there is a ingoing and outgoing transition to a middle element, and the outermost $\max^0$ will return 1 whenever there is such a transition and 0 otherwise. Operator $\longrightarrow$ multiplies quantities, resulting in 1 when the tuple is present in the two relations, ~ preserves the binary quantities, and ^ has the standard definition using `.`. Multiplicity constraints and quantifications only test whether quantities are different from 0. The constraints derived from the signature and field declarations are also preserved in QAlloy. □

Moreover, even without having an explicit **Int** signature we can also show that QAlloy is in fact strictly more expressive than regular Alloy, thus completely superseding it for qualitative and quantitative modelling.

Theorem 3.2 (Alloy/QAlloy expressiveness). *Any Alloy model can be translated into a semantically equivalent QAlloy model.*

Proof. An explicit `Integer` signature with the same bounded semantics as Alloy's **Int** can be defined in QAlloy as follows. Declare signature `Integer` and impose a total order over it using the standard `util/ordering` module. Declare a singleton signature `One` that represents an abstract unit for integers. Then, declare a quantitative field `value` within `Integer` that points to `One`. Impose through a fact that the quantity of `value` of every `Integer` corresponds to its position in the total order. If the scope of **Int** in Alloy is n the scope of the equivalent `Integer` in QAlloy should be 2^n and the value of the `first Integer` in the total order should be set as `first;value` = $(2^{n-1} - 1)$`**` `One`. A macro `toInteger[q]` can be defined that converts any quantity `q` of type `One` back to respective `Integer` atom, taking overflows into consideration. It picks the `Integer` whose `value` is the addition of `first;value` with **sub**`[q,first;value]` modulo 2^n. Any wrap around arithmetic operation over `Integer` can now be defined by first performing the respective (unbounded) operation on the respective `value` and then converting back with `toInteger`. Any **Int** standalone constant `c` in the original Alloy model can be represented by the equivalent `toInteger[c ** One]` in QAlloy. Finally, any summation **sum** `x : A | E` can be replaced by the equivalent `A;{x : A, i : E |` **no none**`};value`. □

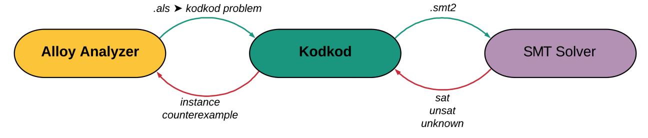

Figure 8: QAlloy workflow

4 QUANTITATIVE ANALYSIS

QAlloy is supported by an Analyzer that extends that of regular Alloy. The overview of the analysis process is depicted in Figure 8 and presented in the rest of this section.

4.1 Quantitative Kodkod

Quantitative Kodkod problems. Kodkod [21] is the relational model finder behind Alloy, and is responsible for verifying the satisfiability of the input problem and further determine an instance solution, when satisfiable. A Kodkod problem is essentially a declaration of relations through lower- and upper-bounds – which specify tuples that must be present or absent in the relation, respectively – and a relational formula that bindings over such relations must guarantee. The Analyzer translates Alloy specifications to Kodkod problems by translating the syntactic features of the language into additional constraints. Kodkod then interprets each declared relation as a Boolean matrix, where each tuple between the lower- and upper-bound is assigned a SAT variable denoting its presence in the relation. Relational operators are computed as matrix operations and formulas expanded to Boolean connectives, with quantifiers unrolled in the finite universe of discourse. The process is highly optimised by resorting to special data-structures to represent the (usually quite sparse) matrices and quantifier-free Boolean formulas.

Achieving quantitative analysis requires dealing with values beyond the Boolean realm, which led to the development of a quantitative Kodkod extension. First, Kodkod's AST was extended to consider quantitative relations and the new operators supported by QAlloy. Then, in order to represent quantitative relations, the underlying *sparse sequence* – a sequence that may or may not have contiguous *flat indices*, storing only non-false values – used to represent Boolean matrices is adapted to support integer values in a similar fashion, now describing *numeric matrices*.

Definition 4.1. Let R be an n-ary *quantitative* relation with lower-bound L and upper-bound U over the universe $\mathcal{A} = \{a_0, a_1, .., a_k\}$. R is uniquely characterized by a numeric sparse matrix specified as follows:

$$R[i_1, .., i_n] = \begin{cases} trueVar() & \text{if } (a_{i_1}, .., a_{i_n}) \in L \\ freshVar() & \text{if } (a_{i_1}, .., a_{i_n}) \in U - L \\ 0 & \text{otherwise} \end{cases}$$

where $i_1, .., i_n \in [0, k]$, $freshVar()$ declares a fresh integer variable, and $trueVar()$ declares a fresh integer variable by additionally imposing that its value must be nonzero.

Such representation means that every n-ary relation is encoded as a matrix of n dimensions, each of size $|\mathcal{A}|$ (e.g., a binary relation is described by a $|\mathcal{A}| \times |\mathcal{A}|$ square matrix). Notice that unlike Kodkod

where a tuple in the lower-bounds is simply assigned the value *true*, in the quantitative setting the presence of a tuple only entails that it will have a nonzero value, but it will still be a free variable. Following the original flat index mechanism [20], an entry $[i_1, .., i_n]$ of R is mapped into the index $\sum_{j=1}^{n}(i_j \times |\mathcal{A}|^{n-j})$ on the respective sparse sequence r. Thus, any tuple over $\mathcal{A}$ is uniquely identified by its index. In this setting, if $r_i = q$ with $q \in \mathbb{Z} \setminus \{0\}$, the tuple corresponding to the flat index i occurs in R with the quantity q associated.

For relations not declared as quantitative, this definition is narrowed to represent Boolean matrices within the $\mathbb{Z}$ domain.

Definition 4.2. Let R be an n-ary *qualitative* relation with lower-bound L and upper-bound U over the universe $\mathcal{A} = \{a_0, a_1, .., a_k\}$. R is uniquely characterized by a numeric sparse matrix specified as follows:

$$R[i_1, .., i_n] = \begin{cases} 1 & \text{if } (a_{i_1}, .., a_{i_n}) \in L \\ \mathit{binaryVar}() & \text{if } (a_{i_1}, .., a_{i_n}) \in U - L \\ 0 & \text{otherwise} \end{cases}$$

where $i_1, .., i_n \in [0, k]$ and $\mathit{binaryVar}()$ declares a fresh integer variable, constrained to be $\{0, 1\}$-valued.

Combining such structures through the existing Boolean operators alongside the expected arithmetic and inequality operations over integers, quantitative properties can be conveniently managed through linear algebra. As an example, consider the implementation of quantitative composition `;`, which is derived directly from the Boolean matrix composition by swapping $\wedge$ by $\times$ and $\vee$ by $+$, respectively, amounting to the multiplication of matrices.

Quantitative analysis. Given a Boolean formula, Kodkod deploys off-the-shelf SAT solvers to find a valuation for the free SAT variables. To support integer variables in quantitative analysis, the process is adapted to integrate SMT solvers instead. Every formula is directly translated into assertions in the *assertion stack* that defines the SMT specification to be fed into an off-the-shelf SMT solver. The generated specifications abide to the SMT-LIB format [3]. Consequently, any solver that conforms to this standard can then be integrated and used by the QAlloy Analyzer. For instance, consider the expression $R = \textbf{add}[S, T]$, where R, S and T are quantitative relational expressions, and R contains the result of performing matrix addition between S and T. Such expression is translated into an SMT specification by pushing assertions of the shape `(assert (=` r_i `(+` s_i t_i`)))` to the stack, for every index i of the sparse sequences, with r_i, s_i and t_i being declared as *integer function symbols*.

SMT solvers perform verification according to a specific background theory. In this context, analysis will be performed with the *Theory of Integers*, in particular, considering the logic fragment `QF_NIA` (*Quantifier-free non-linear integer arithmetic*), the smallest fragment able to cover the kind of function symbols and assertions in the generated specification. After solving, a judgement will be obtained that will be either: *unsatisfiable*, when the solver is unable to find a solution to the model; *unknown* when the solver is unable to reach a conclusion; and *satisfiable* when the solver is able to determine a solution to the SMT specification. In the last case, the SMT variables are interpreted back as a binding over the free relations of the original quantitative Kodkod problem, representing a quantitative instance.

4.2 Optimization

Although quantitative properties are the main focus of this work, qualitative relations and constraints are frequently used even within quantitative models. Therefore, instead of using linear algebra over integer values exclusively, by statically identifying qualitative relational expressions, it is possible to encode some constraints over the Boolean domain without loss in expressiveness, thus reducing the load on the solver. To this purpose, we implement in quantitative Kodkod a simple type inference mechanism that registers whether a relational expression is necessarily qualitative. This is done by registering which relations are not declared as **int**, and how the different relational operators affect this classification. For example, intersecting two qualitative expressions yields a qualitative one, but intersecting qualitative and quantitative expressions yields a quantitative one.

This typing information is then used to choose the best implementation for each operator. First, observe that a binary integer value is exactly described by a Boolean variable, the same kind of value used in Definition 4.2 to characterize qualitative relations. Then, operations closed under the Boolean domain can be described precisely as in the original Kodkod implementation by taking advantage of Boolean constructs. Note that such Boolean values can also be easily lifted into the integer domain when such representation is required. The quantitative Kodkod extension supports both kinds of structure in a way that allows convenient processing of the problem at hand.

Besides the Boolean operators of standard Kodkod, newly added operations, namely those used in numeric expressions, can also be optimized into Boolean expressions. Take for example the Hadamard product between two relational expressions $R = \textbf{mul}[S, T]$. Instead of simply multiplying each entry $r_i = s_i \times t_i$, by observing that $r_i = s_i \wedge t_i$ when S and T represent non-quantitative relational expressions, the implementation of this expression can be optimized as follows, for every index i within the respective sparse sequences:

$$r_i = \begin{cases} s_i \wedge t_i & \text{if } s_i, t_i \in \mathbb{B} \\ s_i \ ?\ t_i : 0 & \text{if } s_i \in \mathbb{B}, t_i \in \mathbb{Z} \\ t_i \ ?\ s_i : 0 & \text{if } s_i \in \mathbb{Z}, t_i \in \mathbb{B} \\ s_i \times t_i & \text{otherwise} \end{cases}$$

Notice that this implementation also optimizes the case where only one of the operands is qualitative.

Finally, at the SMT level, Boolean function symbols and expressions over them will be preferred wherever the expressiveness of their integer counterpart is not required, resulting in faster response times.

4.3 QAlloy Analyzer

The QAlloy Analyzer[5] parses the provided model according to the language extension defined in the previous section, including

[5] QAlloy and all models used in the evaluation are available at https://github.com/pf7/QAlloy.

the typing rules for the new operators, which ensures that the usage of integers is well-typed, as motivated earlier. Moreover, such specification is processed adequately to be handled by the updated relational model finder, i.e., it is transformed into a quantitative Kodkod problem. The constraints generated from the syntactic structures of Alloy when creating the Kodkod formula are preserved by QAlloy (namely, hierarchy and typing constraints), preserving the semantics of non-quantitative signatures and fields. As before, an instance is returned by the quantitative Kodkod analysis when there is an *instance*/*counterexample* to the **run**/**check** command that was executed.

The various components of the Analyzer were adapted to the quantitative context. Namely, the Visualizer and Evaluator now accommodate the quantitative Alloy instances in order to faithfully present the outcome to the designer. For quantitative signatures and fields, the graph perspective of the Visualizer was adapted to display the tuple quantities. For instance: for each subset `s`, the quantity `q` associated with each atom is displayed as `s: q` inside the respective node; the quantity `q` associated with each tuple of a field `f` is displayed in the respective edge as `f: q`. For non-quantitative relations, the Visualizer still follows the original Alloy implementation. This graph visualization is illustrated in Figure 4. To further inspect each obtained instance, the user can also take advantage of the Evaluator by specifying expressions written with the updated language, which will be measured over the quantitative instance in question.

5 EVALUATION

This section presents our evaluation of QAlloy, focusing on the expressiveness of the language and the performance of the Analyzer.

5.1 QAlloy Examples

To evaluate the expressiveness of QAlloy, we have written various examples from distinct domains, inspired by previous attempts at quantitative modelling using regular Alloy[6].

Supermarket self-checkout (`SCO`*).* This is the running example already presented throughout the paper. Quantitative relations are used to model stock quantities and weights. It has two assertions that check whether the quantities of certain items can be inferred from the total weight of the bag: one is invalid (`hasMilk`) and another valid (`hasNoMilk`). Signature scopes do not affect the validity since only the number of bags varies. Previous work has modelled similar shopping bags in Alloy using multi-sets where edges were reified as model atoms [19].

Flow networks (`FlowA` *and* `FlowL`*).* A model of flow networks, directed graphs with limited capacities on the edges and distinct source and sink nodes. The (valid) assertion tests whether the flow produced by the source is the same as that consumed by the sink. Model size determines the number of nodes in the network. There are two different QAlloy models of this example: `FlowA` abstracts the unit of what is flowing through the network, which results in a more cumbersome specification, requiring the usage of operator `#` to ensure type compatibility; `FlowL` explicitly declares the flow in *liters*, which makes the model more elegant, but increases the arity of the affected quantitative relations by one.

Graph analysis (`Graph`*).* A model for analysing quantitative properties of vertex-labelled graphs. Three (valid) assertions verify properties regarding the counting of labels depending of the shape of the graph, namely arbitrary graphs (`Counting`), forest graphs (`ForestInDegree`), and connected graphs (`ConnectedInDegree`). The scope determines the number of nodes in the graph. Alloy has traditionally been used to analyse graph problems, from more abstract problems (such as those packaged with the Analyzer) to more concrete ones (e.g., reasoning about student submissions in automated assessment systems [11]).

Electronic purse (`Bank`*).* A model of an electronic purse and associated transference operations. Quantitative relations are used to model coin quantities. The (valid) check tests whether the total number of coins is always preserved (`Preserves`), whose scope controls the number of banks, clients and transfer orders. Previous work has modelled such electronic purses in Alloy to check security properties [17]. In that work, rather than using integers to model coin quantities, coins were modelled as a signature. This alternative prevents some issues of Alloy's integers, but is still affected by scope problems (the number of available coins) and makes specification more complex (even though coins are fungible, since each coin is a different element of the domain, additional constraints must be imposed to avoid coin sharing, for example).

5.2 Performance Evaluation

This section aims to answer the following research questions:

- **RQ1** How does the performance of QAlloy compare to the bounded integer implementation of regular Alloy?
- **RQ2** Which SMT solver is more efficient in the analysis of quantitative relational models?
- **RQ3** What is the impact of identifying Boolean sub-expressions and exploiting them using the optimization from Section 4.2?

To answer these questions, we executed the commands of the examples under various configurations and scopes. The summary of the results is shown in Table 1. To answer RQ1, we additionally modelled all the examples presented in the previous section in plain Alloy[7]. Here, an additional scope on integers is required, and we executed the models using the MiniSAT solver for scopes that allowed a reasonable number of integers without completely encumbering the Analyzer (columns Alloy in Table 1). For RQ2, we executed the commands with different SMT backends, namely Z3, MathSAT, CVC4 and Yices (columns QAlloy in Table 1). For RQ3, we also ran the commands without the optimizations described in Section 4.2 enabled using MathSAT, the SMT solver with the best overall performance in RQ2 (columns QAlloy-NO in Table 1). All commands were executed on a machine equipped with 8 GB of RAM and an octa-core Intel i7 CPU of 2.5 GHz frequency and x86_64 architecture; both Alloy and QAlloy ran with 768 MB of maximum memory and 16384k of maximum stack size, with Alloy having *prevent overflows* set to *On*. Versions MiniSAT 2.2.1, Z3

[6]Available at https://github.com/pf7/QAlloy/tree/master/org.alloytools.alloy.extra/extra/models/examples/qalloy.

[7]Also available in the GitHub repository.

Table 1: Evaluation results (in ms).

Model	Command	Result	Alloy (MiniSAT) 8 Int	Alloy (MiniSAT) 10 Int	QAlloy Z3	QAlloy MathSAT	QAlloy CVC4	QAlloy Yices	QAlloy-NO MathSAT
SCO	**run**	Sat	253	2650	41	16	39	12	27
	hasMilk	Sat	266	3300	43	19	66	12	176
	hasNoMilk	Unsat	193	2863	47	42	131	17	120
FlowA	**run for** 4 **but exactly** 3 Node	Sat	335	4410	85	49	335	325	40
	OutEqualsIn **for** 3	Unsat	1700	3740	184	33	364	726	79
	OutEqualsIn **for** 4	Unsat	timeout	timeout	13400	108	1800	34700	333
FlowL	**run for** 4 **but exactly** 3 Node	Sat	335	4410	114	46	465	69	54
	OutEqualsIn **for** 3	Unsat	1700	3740	202	28	179	165	49
	OutEqualsIn **for** 4	Unsat	timeout	timeout	10800	66	327	1600	115
	OutEqualsIn **for** 5	Unsat	timeout	timeout	timeout	155	614	timeout	timeout
Graph	**run for** 5	Sat	23	34	209	99	640	38	583
	Counting **for** 3	Unsat	3600	24100	2400	73	35100	timeout	timeout
	Counting **for** 4	Unsat	timeout	timeout	31200	245	timeout	timeout	timeout
	Counting **for** 5	Unsat	timeout	timeout	timeout	893	timeout	timeout	timeout
	run Forest **for** 5	Sat	16	41	144	126	717	43	timeout
	ForestInDegree **for** 3	Unsat	50	279	135	65	290	608	12500
	ForestInDegree **for** 4	Unsat	385	2420	1300	730	2600	timeout	timeout
	ForestInDegree **for** 5	Unsat	2200	15600	29500	22200	46100	timeout	timeout
	run Connected **for** 5	Sat	13	34	401	timeout	1000	50	timeout
	ConnectedInDegree **for** 3	Unsat	61	501	195	158	335	18700	7400
	ConnectedInDegree **for** 4	Unsat	291	2200	1700	31700	2400	timeout	timeout
	ConnectedInDegree **for** 5	Unsat	2400	15500	51900	timeout	27300	timeout	timeout
Bank	**run**	Sat	12	814	68	27	197	105	86
	Preserves **for** 3 **but** 2 Bank	Unsat	4300	timeout	2000	150	3100	timeout	3000
	Preserves **for** 4 **but** 2 Bank	Unsat	15400	timeout	28100	1900	42700	timeout	timeout
	Preserves **for** 5 **but** 2 Bank	Unsat	53000	timeout	timeout	10400	timeout	timeout	timeout

4.8.12, MathSAT 5.6.6, CVC4 1.8, Yices 2.6.4 of the solvers were used. Timeout was set at 1 minute.

Regarding RQ1, QAlloy is competitive with the bounded versions in Alloy. With 8 bits (256 integers), Alloy is already outperformed by QAlloy in three of the examples, while in the other (Graph) it is in the same order of magnitude with 10 bits. Recall that QAlloy has the added advantage of performing the analysis over an unbounded integer domain, while Alloy is limited to the bit-width defined by the user, which can lead to faulty results if not big enough, as discussed in Section 2.

Considering RQ2, although MathSAT features the best overall performance, there is at least one command where each of the different solvers has the best performance. Thus, the QAlloy Analyzer benefits from supporting a wide range of solvers that abide to the SMT-LIB standard. The least performant overall is Yices, with many timeouts, although still competitive when solving **run** commands. Note that the generated SMT problems can be in the QF-NIA fragment which is not decidable. However, we have not experienced this limitation in practice: several commands run in the evaluation are in this fragment, but they are all solved by at least one of the SMT solvers under the 1 minute timeout.

Lastly, for RQ3 we also executed the commands without the optimization described in Section 4.2 with MathSAT. Without this optimization, the tool struggles to find a timely answer for some commands, but with the optimization enabled, commands can execute orders of magnitude faster (e.g., `ForestInDegree` **for** 3 is almost 200x faster with the optimization enabled).

6 RELATED WORK

Previous research has attempted to support (positive) multi-relations in formal modelling, but only through shallow embeddings as library support. For instance, Hayes [8] proposed a binary multi-relations library for Z, implementing the quantitative version of composition (there simply dubbed composition). More closely related to QAlloy, Sun et al. [19] proposed an Alloy library for multi-sets and binary multi-relations. This is based on the category-theoretical framework of spans, where each link is reified as an index (an index-based approach to multi-relations, in contrast to a numeric-based that assigns multiplicities to elements). In practice, links are reified into Alloy atoms, which means they are bounded by the analysis scope. Two composition operators are available, quantitative composition (there dubbed multijoin) and the ordinary Boolean dot join that only deals with reachability and retains a single link. Our dot join operator extends the behaviour of the ordinary join, exhibiting the standard behaviour when restricted to Boolean relations.

Various authors have tried to replace Alloy's SAT-based backend by a SMT-based one. AlloyPE, proposed by El Ghazi et al. [7],

provided the first translation of Alloy's relational logic into SMT. CRS, proposed by Meng et al. [12], puts forward an alternative translation that relies on an extended theory of finite relations. Both allow the analysis of Alloy models without requiring the finitization (i.e., assigning scopes) of signatures (including integers). The evaluation in [12] has shown that CRS, compared to AlloyPE, is able to terminate in more cases and is more robust. Although CRS is able to analyse Alloy models with unbounded integer semantics, the current version does not fully support the cardinality operator (#) nor the **sum** quantifier (the cardinality operator can be used only in comparisons with constants). Unfortunately, this does not allow the encoding of most of our examples, and that was the reason why we did not include the comparison against CRS as a specific research question in our evaluation. Our only example that does not use # nor **sum** is `Bank`. In this example, CRS is much slower than QAlloy: the **run** takes 900 ms and the `check Preserves` more than 4 min (both with unbounded scopes), while in QAlloy with MathSAT the former takes 27 ms (with the default scope 3) and the latter 10 s (with `scope 5 but 2 Bank`). AlleAlle [18] is an SMT-based relational model finder, being at a lower-level of abstraction than Alloy. It is inspired by Codd's relational algebra [5], allowing constraints over data and supporting optimization problems. Constraints may act on unbounded data, namely integers, but declared relations are still bounded. By essentially having the same syntax as Alloy, all these extensions still suffer from the language drawbacks described in Section 2 when dealing with quantitative problems (e.g., lack of dimensional type-safety).

As already mentioned, an alternative integer semantics where overflows are forbidden has been proposed and integrated in the official Alloy Analyzer [13]. It implements a three-valued semantics of arithmetic operations so that overflows are detected and instances where overflows occur removed from the search space. While useful in certain scenarios, it may also give the user a false sense of confidence since relevant counter-examples may be discarded.

Some extensions of Alloy have been proposed to address optimization problems. The already mentioned AlleAlle [18] supports maximization/minimization objectives through the νZ optimizing SMT solver. AlloyMax [22] is able to generate maximal/minimal solutions by relying on PMax-SAT solvers. Alloy* [14], a solver for higher-order relational models can also be used to generate optimal solutions. Quantitative problems requiring optimization (considering not only the number of tuples, as in AlloyMax, but also their quantities) are an interesting class of problems that are still not addressed by QAlloy.

7 CONCLUSION

This paper describes QAlloy, a quantitative relational modelling language based on Alloy, along with its automated Analyzer. Instead of having explicit integers, QAlloy internalizes them in relations, generalizing the Alloy semantics based on Boolean matrices to a linear algebra semantics based on integer matrices. It also replaces the SAT-based analysis of Alloy by an SMT-based one, enabling unbounded quantities. Our evaluation has shown that the approach is feasible, often outperforming the Alloy bounded versions. Alloy has shown to be well-suited to reason about qualitative requirements in structural design. We believe QAlloy also enables the application of its popular features to the analysis of the quantitative requirements that are now ubiquitous in software development.

In the future we intend to continue researching the topic of quantitative relational modelling, namely to improve QAlloy and its Analyzer. First, while the QAlloy Analyzer already supports basic instance iteration, quantities introduce an additional layer of complexity that affects scenario exploration. We intend to explore techniques to provide richer exploration operations, namely operations that force changes in the structure of the instances or meaningful changes in quantities. Second, we intend to identify and formalize a subset of QAlloy that is effectively decidable and guarantees an answer by the SMT backend. Nonetheless, a definitive answer was always returned by at least one of the solvers in all our examples. Related to that, we intend to implement in the QAlloy Analyzer some sort of portfolio solving (either natively or using an off-the-shelf implementation such as Par[8]) precisely to maximize the chances of getting a definitive answer. Third, we intend to experiment with other quantitative domains, for instance, exploring a semantics based on stochastic matrices. This would allow QAlloy to address, for instance, problems related to risk analysis [15]. Finally, we intend to research the integration of QAlloy with the latest version 6 of Alloy [10], that now supports mutable relations and temporal logic to enable behavioural analysis. The challenge here is upgrading the SAT-based model checking backend of Alloy 6 to use SMT solvers.

ACKNOWLEDGMENTS

Our thanks to the anonymous referees for their helpful comments. The work by José N. Oliveira is financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference PTDC/CCI-COM/4280/2021. The work by the remaining authors is financed by National Funds through the FCT within project LA/P/0063/2020.

REFERENCES

[1] K. M. Abadir and J. R. Magnus. 2005. *Matrix algebra*. Econometric exercises, Vol. 1. Cambridge University Press. https://doi.org/10.1017/CBO9780511810800
[2] S. Andova, A. McIver, P. R. D'Argenio, P. J. L. Cuijpers, J. Markovski, C. Morgan, and M. Núñez (Eds.). 2009. *Proceedings of the 1st Workshop on Quantitative Formal Methods: Theory and Applications*. EPTCS, Vol. 13. https://doi.org/10.4204/EPTCS.13
[3] C. Barrett, P. Fontaine, and C. Tinelli. 2016. The satisfiability modulo theories library (SMT-LIB). `www.SMT-LIB.org`.
[4] W. D. Blizard. 1990. Negative membership. *Notre Dame Journal of Formal Logic* 31, 3 (1990), 346–368. https://doi.org/10.1305/ndjfl/1093635499
[5] E. F. Codd. 1970. A relational model of data for large shared data banks. *Communications of the ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685
[6] J. Edwards, D. Jackson, and E. Torlak. 2004. A type system for object models. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 189–199. https://doi.org/10.1145/1029894.1029921
[7] A. A. El Ghazi and M. Taghdiri. 2011. Relational reasoning via SMT solving. In *Proceedings of the 17th International Symposium on Formal Methods (LNCS, Vol. 6664)*. Springer, 133–148. https://doi.org/10.1007/978-3-642-21437-0_12
[8] I. J. Hayes. 1992. Multi-Relations in Z. *Acta Informatica* 29, 1 (1992), 33–62. https://doi.org/10.1007/BF01178565
[9] D. Jackson. 2019. Alloy: A language and tool for exploring software designs. *Communications of the ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

[8] Available at https://github.com/tjark/Par.

[10] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg. 2016. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 373–383. https://doi.org/10.1145/2950290.2950318

[11] N. Macedo, A. Cunha, J. Pereira, R. Carvalho, R. Silva, A. C. R. Paiva, M. S. Ramalho, and D. C. Silva. 2021. Experiences on teaching Alloy with an automated assessment platform. *Science of Computer Programming* 211 (2021), 102690. https://doi.org/10.1016/j.scico.2021.102690

[12] B. Meng, A. Reynolds, C. Tinelli, and C. W. Barrett. 2017. Relational constraint solving in SMT. In *Proceedings of the 26th International Conference on Automated Deduction (LNCS, Vol. 10395)*. Springer, 148–165. https://doi.org/10.1007/978-3-319-63046-5_10

[13] A. Milicevic and D. Jackson. 2014. Preventing arithmetic overflows in Alloy. *Science of Computer Programming* 94 (2014), 203–216. https://doi.org/10.1016/j.scico.2014.05.009

[14] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. 2019. Alloy*: A general-purpose higher-order relational constraint solver. *Formal Methods in System Design* 55, 1 (2019), 1–32. https://doi.org/10.1007/s10703-016-0267-2

[15] D. Murta and J. N. Oliveira. 2015. A study of risk-aware program transformation. *Science of Computer Programming* 110 (2015), 51–77. https://doi.org/10.1016/j.scico.2015.04.008

[16] J. N. Oliveira. 2012. Towards a linear algebra of programming. *Formal Aspects of Computing* 24, 4-6 (2012), 433–458. https://doi.org/10.1007/s00165-012-0240-9

[17] T. Ramananandro. 2008. *Mondex*, an electronic purse: Specification and refinement checks with the *Alloy* model-finding method. *Formal Aspects of Computing* 20, 1 (2008), 21–39. https://doi.org/10.1007/s00165-007-0058-z

[18] J. Stoel, T. van der Storm, and J. 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*. ACM, 46–61. https://doi.org/10.1145/3359591.3359726

[19] P. Sun, Z. Diskin, M. Antkiewicz, and K. Czarnecki. 2016. Modeling and reasoning with multirelations, and their encoding in Alloy. In *Proceedings of the 16th International Workshop on OCL and Textual Modelling (CEUR Workshop Proceedings, Vol. 1756)*. CEUR-WS.org, 73–88.

[20] E. Torlak and D. Jackson. 2006. *The design of a relational engine*. Technical Report MIT-CSAIL-TR-2006-068. MIT.

[21] E. Torlak and D. Jackson. 2007. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 632–647. https://doi.org/10.1007/978-3-540-71209-1_49

[22] C. Zhang, R. Wagner, P. Orvalho, D. Garlan, V. M. Manquinho, R. Martins, and R. Kang. 2021. AlloyMax: Bringing maximum satisfaction to relational specifications. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 155–167. https://doi.org/10.1145/3468264.3468587