On Correctness of Models and Reformulations

Preliminary Version

David Mitchell

Simon Fraser University

Abstract. Generation, reformulation and transformation of models are central activities in the application of constraint modelling languages. Producing correct models and re-formulations is not easy, despite the natural and intuitive declarative semantics of popular languages. A reformulation is a model of a problem that is different from the original, but "essentially the same problem", just "viewed differently". Reformulations are often obtained by applying a transformation to a given model. We consider the problem of ensuring that such a transformation is correct. To treat correctness as mathematically meaningful, we adopt the viewpoint that a constraint modelling language is an enriched logic, and a constraint model defines a class of finite structures. It then becomes practical to prove basic properties related to correctness. As an example, we show that equivalence of models in reasonable modelling languages is not even semi-decidable. We also give evidence that the notion of a reformulation being "essentially the same problem" or not, while intuitively appealing, may have little content. If so, asking if one problem is a correct reformulation of another may not be meaningful. On a more positive note, we show how interpretations and transductions, tools coming from classical model theory and language theory, support formal and implementable notions of correctness of transformations and reformulations.

1 Introduction

Constraint modelling languages, such as MiniZinc [12] and Essence′ [14], along with the software systems that support them, provide useful tools for solving a variety of computational search and optimization problems. Implemented solvers using these languages, such as the MiniZinc system[8] and the the Conjure-Saville Row-Minion tool suite[3], are effective in practice for a growing variety of problems. In principle, the user of such a "model-and-solve" system simply describes their problem in a chosen "high-level" language and submits the description – or "model" – together with a problem instance to the solver. The solver then constructs a solution for the instance and returns it, or reports that there is no solution.

Reality is less simple. Writing a correct model is often not easy. Moreover, the performance of solvers varies greatly with the modelling choices that are made. Users – even those with considerable practical experience – tackling non-trivial instance distributions often must experiment with many modelling choices to

obtain good performance. If getting one model correct is not easy, presumably getting many models correct in a short period of time while simultaneously trying to experimentally evaluate their performance — itself a nontrivial exercise — will be harder.

In practice, confidence in models or specifications in these languages, and the tools that support them, derives largely from their having an intuitive informal declarative semantics. This similar to general programming practice, where confidence in correctness derives primarily from intuitive understanding of language semantics plus whatever degree of testing the developer finds appropriate. This may be adequate up to a point, but as the tools are able to solve a larger variety of problems and are applied in a wider range of settings, precise and verifiable notions of correctness will become valuable. We see at least two reasons this should be the case.

First, the software systems that use constraint modelling languages will continue to become more complex. To aid users in effective problem solving, systems will automatically employ a range of transformations to models (see, e.g., [13]). Some of these transformations, and the code that implements them, will be carefully analyzed and tested. Over time, however, their number, variety and complexity will increase, and they will be composed and modified in complex ways which were not anticipated at design time. Some will be generated automatically based on products of data mining or machine learning, or via mixed-initiative methods with spontaneous human input during real modelling and solving activities. Testing alone cannot ensure correctness in such a setting.

Second, in critical applications involving complex problems with hard instance distributions, we will need ways to verify correctness of models, including methods of performing transformations on models, that can be formally verified. We ultimately would like software tools that go beyond helping a modeller create models that result in good run-times, and automatically produce formally check-able witnesses of correctness of models, of the transformations we apply to them, and of the solutions (or claims there are no solutions) that are output.

Doing so requires a suitable formal notion of the semantics of the modelling languages involved, preferably using a method that is uniform across all languages. We would like such a formalization to, at the least, address a few fundamental issues: what does a constraint modelling language do, when are two models equivalent, and how can we ensure that a transformation applied to a model produces an equivalent model? We also want these verification operations, to the extent possible, to be automatable in software.

Two models both correct if they are, in some appropriate sense, semantically equivalent. But in what sense? Various modelling languages have their semantics specified in different ways (and sometimes not at all). A model could be viewed, for example, as specifying a problem; or as specifying a set of simple constraints; or as specifying a computation that maps input data to a set of simple constraints, etc.

We take the position here that, whatever notion of semantics a language designer might have had in mind, constraint models in essentially all existing

languages can be viewed as defining classes of finite structures, in the sense that term is used in logic. This may not address all problems of correctness, but allows us to make a useful start in a rigorous and language-independent way. For example, this view makes it straightforward to prove that, for essentially all existing constraint modelling languages, equivalence of models is undecidable.

Reformulation is (roughly speaking) the construction of alternate models of a problem. We focus on the case where the two models use different sorts of variables, or in linguistic terms, different vocabularies. For example, we can write model for the n-Queens problem using a 2-d array that represents all board squares, or using a 1-d array that identifies, for each column, which row has a queen in that column. For two models over identical vocabularies, equivalence is a relatively clear notion. For two models over different vocabularies the very notion of "being essentially the same problem" is tricky. Existing results on special reductions between NP-complete problems suggest strongly that there may be no practically useful formal notion of "being the same problem".

However, if we take a different tack, there are tools that can help us. In classical model theory an interpretation is a map between two vocabularies (or between structures for those vocabularies), that lets us translate properties or queries expressed in one vocabulary into properties expressed in the other vocabulary. An interpretation applied "in reverse" is a map from one class of structures to another, sometimes called a transduction (generalizing the use of that term in language theory, see e.g., [4]). Re-formulation activities frequently involve applying operations that map one model to another. Formally justified testing and verification schemes can be obtained by associating transductions with transformations used in model reformulation.

The remainder of the paper is organized around the four previous points, as follows. (We don't call these "results", as the first argues for adopting a point of view, and the other three are rather immediate consequences of doing so.)

1. In Section 2, we briefly illustrate that constraint models can be viewed as defining classes of finite structures. It is not hard to show that most features of real constraint modelling languages can be seen semantically as moderate, often conservative, extensions of classical first order logic. Some of the issues involved are addressed in [11, 15, 9].
2. In Section 3, we show that, for most reasonable constraint modelling languages, the equivalence relation on models is not recursively enumerable, so there is not even a semi-decision procedure for showing two models are equivalent.
3. In Section 4), we point out that existing results, known as isomorphism theorems for NP, suggest there is no satisfactory formal version of the informal notion that two models with distinct vocabularies define "essentially the same problem".
4. In Section 5, we show that logically defined transformations on structures can be associated with transformations used in reformulation activities, and these can be employed to produce useful tests, witnesses and correctness proofs.

2 Models, Structures and Transductions

Each model or specification in a constraint modelling language contains a collection of declarations of data objects and a collection of constraint expressions. Other elements that appear in some models are primarily solver directives of one sort or another, and not really part of the problem model.

Consider the following simple MiniZinc example, for the problem of colouring a map of the states and territories of Australia:

```
EX1 ≡
enum Color;
enum State = {sa,q,nsw,v,wa,nt,t};
array[States] of var Color: Map ;
constraint
   forall(i in States)(
      forall(j in States)
         ( if i != j then Map[i] != Map[j] )
   );
```

The model, called `EX1`, declares three data objects. `Color` and `State` are declared as type `enum`, so they are (ordered) sets of abstract objects. `Map` is an array with indices in `State` and elements in `Color`, so it is a function mapping states to colours. The constraint ensures that any two distinct states are mapped to different colours. (We simplified the usual requirement that only neighbouring states are differently coloured to keep the model small.) The model is of a search problem that we might describe as follows:

```
Instance: Set Color, of colours;
Solution: Function Map, from the set $\{$sa,q,nsw,v,wa,nt,t$\}$
   of states to Color, such that no two states are mapped to
   the same colour.
```

The `var` modifier in the declaration of `Map` tells us that the solution to the problem is a denotation for `Map`. The assignment of `sa,q,nsw,v,wa,nt,t` to `State` tells us that `State` is a constant. The absence of either for `Color` tells us that `Color` is part of the input. Ignoring these distinctions, we have three symbols that denote data objects: `Color`, `State`, and `Map`. Given a problem instance (a denotation for the symbol `Color`), a solution is a denotation for the symbol `Map` such that the three objects together satisfy all given constraints.

Every constraint model has this nature: a collection of declarations of specific data objects and a collection of constraints. The collections of data objects used in constraint models vary widely, but in all cases they consist of a finite collection of finite functions and relations of particular arities. Each function or relation has a name that is used in the constraints. In mathematical logic, such a collection is called a finite structure, and the names and arities are the vocabulary for that structure.

Remark 1. For our purposes, there is no useful distinction between "specification" and "model". Therefore, since we also must use the term "model" with its meaning in logic — a structure that satisfies a formula — we henceforth use the term "specification" for what most people in the constraints community would call a model.

Let L be a constraint modelling language. An L-specification S involves two sorts of vocabulary (function and relation) symbols: instance symbols (e.g., unary relation symbol `Colour`), and "solution" symbols (e.g., unary function symbol `Map`). The task of the solver is: given denotations for the instance symbols, find denotations for the "solution" symbols that satsify S. For a decision problem, the "solution" vocabulary may be empty, or may contain symbols the solver must find denotations for in order to reach the correct decision. For an optimization problem, there is also a cost function mapping solutions to numerical values. For our narrow purposes in this paper, we do not care much about these roles, but for simplicity and uniformity will write in terms of search problems.

Definition 1 (Vocabulary, Structure). *A vocabulary (or signature) is a tuple of relation and function symbols, each with an associated arity. A structure $\mathcal{A}$ for a vocabulary σ of cardinality r is a tuple of cardinality $r+1$ consisting of a set A, called the universe or domain of $\mathcal{A}$, and denotations for each function and relation symbol of σ. More precisely, for each k-ary relation symbol R in σ, $\mathcal{A}$ contains a k-ary relation over A, denoted $R^{\mathcal{A}}$, and for each k-ary function symbol f in σ, $\mathcal{A}$ contains a k-ary function $f^{\mathcal{A}} : A^k \mapsto A$. $\mathcal{A}$ is called finite if its universe A is finite. We write Struc$[\sigma]$ for the class of all finite σ-structures.*

Let S be a specification with a vocabulary σ for its data objects. If we have a denotation (i.e., a concrete data object) for each symbol, the tuple of these denotations is a finite σ-structure. Typically, it is convenient and works fine to take the universe A to simply be the collection of all elements that appear in any of the data objects. (In database theory, this set is called the "active domain".) If the elements of this structure satisfy all the constraints of S, then the structure consists of a problem instance together with a solution for that instance (and possibly some other things). We can say that S is satisfied by exactly the σ-structures that satisfy all the constraints of S. Alternately, we can say that S defines the class of σ-structures that consist of a problem instance together with a solution for that instance.

In the remainder, we assume that the reader is familiar with classical first order logic (FO), including the recursive definition the binary satisfaction relation $\models$ over structures and formulas. A sentence is a formula with non free variables. A model of a sentence is a structure that satisfies the sentence, and we henceforth use the term "model" in this sense, unless we explicitly say "constraint model".

For a first order formula ϕ, we denote by free(ϕ) the set of free variables in ϕ, and write $\phi(\bar{x})$ to indicate that the free variables of ϕ are among those in tuple $\bar{x}$. If $\mathcal{A}$ is a τ-structure, $\bar{a} \in A^k$ and $\phi(\bar{x})$ a τ-formula with k free variables $\bar{x}$, we write $\mathcal{A}, \bar{a} \models \phi(\bar{x})$ to say that if the variables $\bar{x}$ in ϕ denote the elements of

$\bar{a} \in A^k$ then ϕ is true in $\mathcal{A}$. We write $\phi(\bar{x})^{\mathcal{A}}$, or just $\phi^{\mathcal{A}}$, for the relation defined by ϕ in $\mathcal{A}$. That is, if ϕ has k free variables, then

$$\phi^{\mathcal{A}} = \phi(\bar{x})^{\mathcal{A}} = \{\bar{a} \in A^k \mid \mathcal{A}, \bar{a} \models \phi(\bar{x})\}.$$

We write $\mathrm{Mod}(\phi)$ for the class of all finite models of a sentence ϕ.

Let $\tau = (R_1, \ldots R_m)$ be a vocabulary, $\mathcal{A} = (A, R_1^{\mathcal{A}}, \ldots R_m^{\mathcal{A}})$ a τ-structure, and $(S_1, \ldots S_n)$ a tuple of relation symbols not in τ. If $\mathcal{B} = (A, R_1^{\mathcal{A}}, \ldots R_m^{\mathcal{A}}, S_1^{\mathcal{B}}, \ldots S_n^{\mathcal{B}})$ is a structure for $\tau' = (R_1, \ldots R_m, S_1, \ldots S_n)$, then we call $\mathcal{A}$ the τ-reduct of $\mathcal{B}$ and $\mathcal{B}$ an expansion of $\mathcal{A}$ to τ'.

2.1 Specification Languages

Under the view adopted here, specifications or constraint models contain definitions of classes of finite structures, plus perhaps additional information for a solver to use. In this section, we first give a minimal formal definition of a specification language reflecting that part of constraint modelling languages that define classes of structures.

Definition 2. *A specification language L, is a decidable set of strings imbued with a semantics under which each string $S \in L$ defines an isomorphism-closed class of finite structures. That is, a set of strings L together with a satisfaction relation $\models_L$. The satisfaction relation is a binary relation $\models_L \subset L \times \mathrm{struct}$, where* struct *is the class of all finite structures, and such that for any L-specification S and any two isomorphic structures $\mathcal{A}, \mathcal{B} \in \mathrm{struct}$, $\mathcal{A} \models_L S \Leftrightarrow \mathcal{B} \models_L S$.*

Obviously, for a language L to be interesting or useful it must satisfy many other properties. For example, it should be uniquely parsable and have a semantics that is compositional and truth-functional. But Definition 2 is sufficient for present needs.

By adopting this view, we are giving the language L a model-theoretic semantics, even in the case the language may be viewed as having a different style of semantics in other contexts. Since strings of L define classes of finite structures, they are formulas of some logic, whether they are syntactically like any familiar logic or not.

Remark 2. Constraint modelling languages use additional features. For example, models for optimization problems contain expressions indicating this, and some language have expressions for solver directives. The latter are not really part of the problem model. The former are relevant and interesting, but we set them aside for the present, noting only that the cost function essentially gives a partial order to the structures that contain a solution.

2.2 Translation Schemes and Transductions

In the following sections, we will make use of maps between vocabularies and structures defined by of tuples of first order formulas. We provide some basic definitions here. Our terminology approximately follows [7].

Definition 3 (FO Translation Scheme). *Let τ and $\sigma = (R_1, \ldots, R_m)$ be two relational vocabularies and $C = \{c_1, \ldots\}$ a finite set of constant symbols not in τ or σ. Let Φ be a tuple $\Phi = (\phi_0, \phi_1, \ldots, \phi_m)$ of $|\sigma| + 1$ FO formulas over vocabulary $\tau \cup C$, where the special constants $c_i \in C$ occur only in atoms of the form $x = c_i$. Further, suppose that ϕ_0 has exactly k distinct free variables, and for each relation symbol $R_i \in \sigma$, the number of distinct free variables of the corresponding formula ϕ_i in Φ is exactly $k \cdot \mathrm{ar}(R_i)$. Then Φ is a k-ary τ-σ translation scheme.*

We illustrate with a very simple example, in which k is 1.

Example 1. Let $\tau = (E)$ and $\sigma = (F)$, where E, F are both binary relation symbols. Further, let $\Phi = (\phi_0, \phi_1)$ where $\phi_0(x)$ is $(x = x)$ and $\phi_1(x, y)$ is $(E(x, y) \vee E(y, x))$ Then Φ is a unary $\tau - \sigma$ tranlation scheme.

A τ-σ translation scheme Φ defines two functions. One is a (partial) map from τ-structures to σ-structures. This map is often called a *transduction*, generalising the use of the term in formal language theory. We will define this function formally in this section. The other function is a map from σ-formulas to τ-formulas that lets us answer a query about a τ-structure by translating into a query about a σ-structure. That function will be defined in Section 5.

Before giving definitions, we illustrate these two functions with a simple, if perhaps uninspiring, example.

Example 2. Let Φ be the translation scheme of Example 1, and $\mathcal{A} = (A, E^{\mathcal{A}})$ a τ-structure. We may think of $\mathcal{A}$ as being a directed graph with vertex set A and directed edges as given by $E^{\dashv}$. Now, let $\mathcal{B}$ be a σ-structure defined as follows. The universe of B will be the set $\phi_0^{\mathcal{A}}$, and the relation $F^{\mathcal{B}}$ will be the relation $\phi_1^{\mathcal{A}}$. So, B will be the same as A, and a pair (i, j) will appear in $F^{\mathcal{B}}$ if and only if either (i, j) or (j, i) appears in $E^{\mathcal{A}}$. In other words, $\mathcal{B}$ is the undirected graph underlying the directed graph $\mathcal{A}$. So, we can think of Φ as mapping an a directed graph to its underlying undirected graph. On the other hand, suppose $\mathcal{A}$ is a large directed graph and we want to test a property of it's underlying undirected graph without explicitly generating the second graph. Suppose the property is having a (undirected) path of length 2. The formula $\psi = \exists x \exists y \exists z (F(x, y) \wedge F(y, x) \wedge x \neq z)$ is true of an undirected graph with edge relation F if and only if that graph has a P_2. Now, replace each occurrence of F in ψ with the formula ϕ_1 (with appropriate variable substitutions). We get a new formula $\psi' = \exists x \exists y \exists z ((E(x, y) \vee E(y, x)) \wedge (E(y, z) \vee E(z, y)) \wedge x \neq z)$. Now, ψ' is true of a directed graph with edge relation E if and only if its underlying undirected graph has a (undirected) P_2. So, we can use Φ either as a map taking a directed graph to its underlying undirected graph, or as a map taking a formula defining a certain sort of undirected graph to a formula that defines the directed graphs whose underlying undirected graphs have that property.

Definition 4 (Transduction $\overrightarrow{\Phi}$). *Let $\sigma = (R_1, \ldots, R_m)$ be a relational vocabulary and $\Phi = (\phi_0, \phi_1, \ldots, \phi_m)$ be a k-ary τ-σ translation scheme. Then the transduction $\overrightarrow{\Phi}$ is the partial function from τ-structures to σ-structures defined*

as follows. If $\mathcal{A}$ is a τ-structure and $\phi_0^{\mathcal{A}}$ is not empty, then $\mathcal{B} = \overrightarrow{\Phi}(\mathcal{A})$ is the σ-structure with

- *universe* $B = \{\bar{a} \in (A \cup \{c_i\})^k \mid \mathcal{A}, \bar{a} \models \phi_0\}$;
- *for each* $i \in [1, m]$, $R_i^{\mathcal{B}} = \{\bar{a_1}, \ldots, \bar{a_i} \mid \mathcal{A}, \bar{a_1}, \ldots, \bar{a_i} \models \phi_i\}$

If $k > 1$, the universe of $\mathcal{B}$ is a set of tuples of elements of A and $\{c_i\}$, so a τ-formula that defines an r-ary relation in $\mathcal{B}$ has kr free variables. As an aid to reading, we may denote the k-tuples that make up elements of B with $\langle\rangle$.

Some examples of well-known "reformulations" that are FO transductions are illustrated in [10]:

- Tseitin's linear-time translation of propositional formulas to CNF formulas is a FO transduction.
- The usual linear-time translation of CNF formulas to 3-CNF formulas is a FO transduction.
- Almost all model-and-solve systems take as input a pair consisting of a specification S and a problem instance I, and as a step in solving map this pair to a single expression F in a simple "flat" language, for example a FLatZinc expression or a propositional CNF formula. For each specification S, in a language with at most the expressive power of FO, the flattening function that maps I to F is a FO transduction.

3 Equivalence

The most fundamental question we might ask about a specification S is whether or not it is correct. The computational problem most relevant to correctness is equivalence: Is S equivalent to another given specification T? If S and T are written in an expressive language, then there is no algorithm to test this. To be a little more precise, for any reasonably expressive specification language L, the equivalence relation on L-specifications is not recursively enumerable. Thus, there is no algorithm for checking equivalence of specifications that always eventually terminates on an equivalent pair and always reports a correct answer when it terminates.

Definition 5. *For any specification language L, let $\equiv_L$ be the equivalence relation on L-specifications. That is:*

$$\equiv_L \;=\; \{\langle \phi, \psi \rangle \mid \phi, \psi \in L,\ \mathrm{vocab}(\phi) = \mathrm{vocab}(\psi),\ \mathrm{Mod}(\phi) = \mathrm{Mod}(\psi)\}.$$

Here is a semi-decision procedure for checking non-equivalence of a pair (S_1, S_2) of L-specifications over the same vocabulary σ: Enumerate finite σ-structures, and stop and report "not equvialent" if a structure is found that satisfies one specification but not the other. So $\equiv_L$ is co-re. However, as we will see (and not surprisingly), $\equiv_L$ is not r.e.

Let FO(R) denote the fragment of FO consisting of all function-free formulas in which the only relation symbols are the binary relation symbols R and $=$.

Theorem 1 (Trakhtenbrot[16]). *Satisfiability in the finite for formulas of FO(R) is undecidable.*

That is, given a formula of FO(R), it is undecidable whether the formula has a finite model or not.

Remark 3. An unsatisfiable specification in practice is often one with a bug, since it defines the empty class of structures.

Since most constraint modelling languages are intended to solve (at least) a variety of NP-hard problems, it should not be surprising that they are more expressive than FO(R), which can express only (a proper subset of the) polynomial-time problems. We say there is a (Turing many-to-one) reduction from FO(R) to L if there is a computable function f that maps each formula ϕ of FO(R) to a specification of $S = f(\phi)$ of L with the following properties:

1. S has a data object R that represents a binary relation (that is, either is a binary relation or is an array or function that can represent a binary relation, such as a binary function or a 2-d array that represents the characteristic function);
2. For every finite structure $\mathcal{A}$ for the vocabulary consisting only of R, $\mathcal{A} \models \phi$ if and only if there is a structure $\mathcal{B}$ for the vocabulary of S such that $\mathcal{B} \models \mathcal{S}$ and $R^{\mathcal{B}} = R^{\mathcal{A}}$.

Clearly, satisfiability is undecidable for any specification language for which there is such a reduction from FO(R). It is easy to establish this property for languages that are commonly used in the constraints community, such as MiniZinc, Essence, and others. To do so, it is sufficient to show that the language has a suitable representation of binary relations, and that for each formula of FO(R), we can construct a specification in L that ensures R is defined appropriately. This needs only that the language for expressing constraints can simulate a complete set of Boolean connectives and quantification over a set that is part of the input.

Theorem 2. *If there is a many-one Turing reduction from* FO(R) *to specification language L, then $\equiv_L$ is not recursively enumerable.*

Theorem 2 is a straightforward corollary of Trahktenbrot's Theorem[16].

Proof. Satisfiability of specifications in L is not decidable, because satisfiability for FO(R) is undecidable and is reducible to satisfiability for L. It follows that the set of valid specifications (those that are true in every structure) of L is not r.e., because if a set and its complement are both r.e., then they are also both decidable. Let V be any valid L-specification. To decide if a specification S is valid, we need only decide if $\langle S, V\rangle \in \equiv_L$. This is a reduction from validity to membership in $\equiv_L$. Therefore $\equiv_L$ is not r.e.

Remark 4. A valid specification corresponds either to a decision problem with no "no"-instances, or to a search problem for which every candidate solution is

an actual solution. In practice, such a specification will often be one with a bug. It could, however, be a correct specification for an optimization problem, since there are optimization problems in which every candidate solution is feasible.

We can, of course, often use theorem-proving technology to show that two specifications are equivalent. If we can prove that $\phi \models \psi$ and that $\psi \models \psi$, then we know that $\phi \equiv \psi$. But there are two ways in which we might fail. The first is that, since our languages are expressive, entailment is undecidable and we might fail to find a proof. The second is that we are interested in equivalence over finite models only. We could also have a case in which ϕ and ψ, are equivalent on finite structures (the case we care about), but are not equivalent on infinite structures. In this case, the theorem prover will fail.

4 Reformulation and Sameness

Suppose we have two different formulations or constraint models of a problem, expressed as specifications S and T, where the vocabularies of S and T are different. We would like to ask whether S and T are both correct reformulations of the same problem. When we think they are, we often say that the problems defined by S and T are "essentially the same problem". For example, two different ways of modelling the n-Queens problem, as mentioned previously, have different vocabularies but seem to model "the same problem". The constraints literature contains many (often more interesting) examples. This concept lies at the heart of modelling and reformulation activities, whether performed by a human or by software.

Remark 5. A vocabulary is just a collection of symbols and associated arities. Obviously, it is not the symbols *per se* that is important: it does not matter if we call the edge relation of a graph E or *Edge* or *Bob*. It is the choice of arities of the relations and *what we intend them to represent* that matters. For a relation symbol R we call this the "intended or intuitive interpretation of R".

It is clear that, in this setting, $\mathrm{Mod}(S) \neq \mathrm{Mod}(T)$: the two classes of structures are not the same, so S and T seem to define different problems. What makes us feel they are "really the same problem"? Is there any way to formalize such an intuition?

Reductions Intuitively, the way we typically think of "sameness" of two problems A and B, is that for each instance I of A: there is a corresponding instance of J of B; that J is easy to construct from I; and that a solution to I is easily obtained from a solution to J. Further, there should be a similar correspondence in the reverse direction.

What we have just described, of course, is the property that problems A and B are reducible to each other. If our requirement for "easy to construct" is

existence of a polytime algorithm, then we are talking about poly-time many-to-one reductions. However, every two NP-complete problems are polytime many-one reducible to each other, so adopting this notion of "sameness" would make all NP-complete problems the same.

This seems much to generous to be a useful notion of sameness. Assuming the initial intuition is about right, it seems we would need to ask for a stronger notion of similarity, or a weaker notion of reduction, or both.

Isomorphism Suppose that we that require S and T are not only reducible to each other, but are isomorphic. This seems quite a strong notion of similarity. The Berman-Hartmanis Conjecture[1] is that all NP-complete languages are p-isomorphic. That is, for every two NP-complete languages L_1,L_2, there is a polytime computable isomorphism between L_1 and L_2. (A polytime isomorphism is, of course, a special polytime many-to-one reduction.) The conjecture was motivated by the observation that all known NP-complete problems at time were p-isomorphic. There is sufficient evidence against the conjecture that many complexity theorists now believe it is false. At the same time, there is stronger evidence that it is in a sense "almost true" – that natural problems which are exceptions are rare or unusual.

FO Reductions and Isomorphisms For reductions, we want somehow to allow only "simple" reductions, so a notion of reduction that is much weaker than polytime computable, so only relatively simple reductions are allowed. But we also need a notion that is broad enough to apply to a wide range of problems. A FO reduction is a FO transduction that is also a reduction. FO reductions are quite weak. For example, no FO formula can define parity or connectivity or transitive closure, so no FO reduction can rely on checking any of these properties. But, we will consider an even weaker kind of reduction. A reduction f is a projection reduction if each bit in $f(x)$ depends on *at most one* bit in x. A FO projection reduction, or fop, is a FO reduction that is also a projection reduction. This would seem to be a very weak family of reductions. Each point in the structure $f(\mathcal{A})$ may depend on at most one point in the structure $\mathcal{A}$. This means, for example, that in a reduction f from one graph problem to another, existence of an edge in $f(G)$ cannot be conditional on a pair of edges in G. Despite this, we still have the following two facts (see, e.g., [5]):

1. SAT is complete for NP under fops, i.e., every problem in NP has a FO projection reduction to SAT;
2. Every two problems that are complete for NP under fops are isomorphic under a FO-definable isomorphism.

There are NP-complete problems that are not complete for NP under fops[5], and it seems likely that there are infinitely many p-isomorphism classes in NP[6]. So, these results do not strongly indicate that all NP-complete problems are p-isomorphic. What they do suggest is that the large majority of the real problems

you will encounter are FO-isomorphic. Therefore, to distinguish between problems that are "essentially the same" or not, we need something much more fine-grained.

It could turn out that the informal notion of two problems being "really the same problem", while intuitively appealing, is just handwaving: we say problems are really the same problem when there is an reduction that is simple in an intuitively appealing manner. For help with rigourous notions of correctness, we need more.

5 Verifiable Transformations

We have that "correctness" is not even semi-decidable, and the the notion of "being the same problem" seems problematic. Yet, reformulation seems a necessary activity for practical applicability. What can we do to formally ensure correctness of reformulations of critical problems?

If we have a method for mapping a specification S over vocabulary σ to a "reformulation" with specification T over vocabulary τ, applying this reformulation involves two functions: one function, say f, maps specifications, that is $f(S) = T$. The second function, say g, maps problem instances from one vocabulary to the other. These two maps must correspond, which means they must preserve satisfaction:

$$\mathcal{A} \models S \;\Leftrightarrow\; g(\mathcal{A}) \models f(S).$$

For a search or optimization problem, unless we can prove this correspondence holds for our implementations of f and g, the best we can do to ensure correctness is to verify it empirically for a small set of test instances. We would like to do more.

One way is to construct our reformulation operations by means that ensure formal verifiability. Here, we will make use of both functions defined by a translation scheme: transductions, as defined previously, and "reverse translation", which we define now.

Definition 6 ("Reverse" Translation $\overleftarrow{\Phi}$). *Let $\sigma = (R_1, \ldots R_m)$ and $\Phi = (\phi_0, \phi_1, \ldots \phi_m)$ be a k-ary τ-σ translation scheme. The translation $\overleftarrow{\Phi}$ is a function from σ-formulas to τ-formulas. We obtain τ-formula $\overleftarrow{\Phi}(\psi)$ from σ-formula ψ by:*

1. *Replacing each atom $R_i(x_1, \ldots, x_m)$ with $(\wedge_j \phi_0(\bar{x_j}) \wedge \phi_i(\bar{x}_1, \ldots \bar{x}_m))$, where each $\bar{x_j} = (x_{j,1}, \ldots x_{j,k})$ is a k-tuple of new variables;*
2. *Replacing each existentially quantified subformula $\exists y \psi$ with $\exists \bar{y}(\phi_0(\bar{y}) \wedge \psi)$, where $\bar{y}$ is a k-tuple of new variables. Universally quantified subformulas are relativized in the dual manner.*

A fundamental property of translation schemes (standard in expositions of model theory) relates their dual role defining translations and transductions.

Theorem 3 (Fundamental Property of Translation Schemes). *Let Φ be a k-ary τ-σ translation scheme. If $\mathcal{A}$ is a τ-structure for which $\overrightarrow{\Phi}(\mathcal{A})$ is defined, and θ is a σ-formula with r free variables $\bar{x}$, then*

$$\mathcal{A} \models \overleftarrow{\Phi}(\theta)(\bar{y_1}, \dots \bar{y_r}) \Leftrightarrow \overrightarrow{\Phi}(\mathcal{A}) \models \theta(x_1, \dots x_r)$$

where $\bar{y_i}$ is the k-tuple of variables corresponding to x_i in the computation of $\overrightarrow{\Phi}$.

Suppose we can associate, to each reformulation rule that takes a specification S to a specification T, a translation scheme Φ such that the transduction $\overrightarrow{\Phi}$ maps σ-structures to τ-structures in correspondence with the reformulation rule. The transduction $\overrightarrow{\Phi}$ is the function g above. We still also require a function f that correctly maps specification S to specification T. (Examples of reformulations actually implemented by FO transductions in a solver can be found in [2].)

Having this translation scheme gives us several additional methods for checking correctness. Here are some examples.

1. If f is correct, then the reverse translation ensures that
$$\mathcal{A} \models S \Leftrightarrow \mathcal{A} \models \overleftarrow{\Phi}(f(S)))$$
which we can test on various instances $\mathcal{A}$, and also that
$$S \equiv_L \overleftarrow{\Phi}(f(S)),$$
from which we might be able to establish correctness of f by theorem proving.
2. A natural way to develop tests for a constraint solving system is to express properties that solutions of a problem must have, but which are not explicit in the specification. In the case of reformulation, we may express such a property independently for the two versions of the problem, or indeed we may have a property which we only understand for the reformulated version. If ϕ is a property that must hold over every solution to the reformulated version, then $\overleftarrow{\Phi}(\phi)$ must hold for every solution to the original problem.
3. When solving problems after complex re-formulation, we may need to map solutions back to the original vocabulary. This requires an additional function h. If h is a FO transduction then we have the following:
$$\mathcal{A} \models \mathcal{S} \leftrightarrow \overrightarrow{\Phi}(\mathcal{A}) \models \overleftarrow{h}(\mathcal{S}).$$
In other words, $\overleftarrow{h}$ maps our specification S to a specification S' such that, if everything is correct, $f(S) \equiv_L S'$.

References

1. Berman, L., Hartmanis, J.: On isomorphisms and density of np and other complete sets. SIAM Journal on Computing **6**(2), 305–322 (1977)
2. Bogaerts, B., Jansen, J., de Cat, B., Janssens, G., Bruynooghe, M., Denecker, M.: Bootstrapping inference in the IDP knowledge base system. New Generation Comput. **34**(3), 193–220 (2016)

3. Conjure, Saville Row, Minion Constraint Modelling Toolchain. https://www.constraintmodelling.org
4. Courcelle, B., Engelfriet, J.: Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach, Encyclopedia of mathematics and its applications, vol. 138. Cambridge University Press (2012)
5. Immerman, N.: Descriptive complexity. Graduate texts in computer science, Springer (1999)
6. Mahaney, S.R.: On the number of p-isomorphism classes of np-complete sets. In: 22nd Annual Symposium on Foundations of Computer Science (FOCS). pp. 271–278 (1981)
7. Makowsky, J.A.: Algorithmic uses of the Feferman-Vaught theorem. Ann. Pure Appl. Logic **126**(1-3), 159–213 (2004)
8. MiniZinc. https://www.minizinc.org
9. Mitchell, D.: Notes on satisfiability-based problem solving. https://www.cs.sfu.ca/~mitchell/satisfiability-based-solving/.
10. Mitchell, D.: Guarded constraint models define treewidth preserving reductions. In: Schiex, T., de Givry, S. (eds.) Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11802, pp. 350–365. Springer (2019). https://doi.org/10.1007/978-3-030-30048-7_21, https://doi.org/10.1007/978-3-030-30048-7_21
11. Mitchell, D.G., Ternovska, E.: Expressive power and abstraction in essence. Constraints An Int. J. **13**(3), 343–384 (2008). https://doi.org/10.1007/s10601-008-9050-3, https://doi.org/10.1007/s10601-008-9050-3
12. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. pp. 529–543 (2007)
13. Nightingale, P., Özgür Akgün, Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in savile row. Artificial Intelligence **251**, 35 – 61 (2017). https://doi.org/https://doi.org/10.1016/j.artint.2017.07.001, http://www.sciencedirect.com/science/article/pii/S0004370217300747
14. Nightingale, P., Rendl, A.: Essence' description. CoRR **abs/1601.02865** (2016), http://arxiv.org/abs/1601.02865
15. Ternovska, E., Mitchell, D.G.: Declarative programming of search problems with built-in arithmetic. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 942–947 (2009), http://ijcai.org/Proceedings/09/Papers/160.pdf
16. Trakhtenbrot, B.: The impossibility of an algorithm for the decidability problem on finite classes. Doklady AN SSR **70**(4), 569–572 (1950)