Noname manuscript No.
(will be inserted by the editor)

Automated Code Generation

an approach using Second-Order Logic

Emmanouil G. Krasanakis ·
Andreas L. Symeonidis (supervisor)

08/08/2015

Contents

1 Introduction

As more programming code becomes available through ever-growing repositories, we steadily approach the point where, for most problems, similar implementations already exist. Thus, in order to save development time, there is a need for methods that re-use already existing code for solving similar problems.

Ideally, such methods infer the required code from user specifications. Towards that end, in this paper we try to approach the problem through Second-Order Logic, which can model both programming languages and free speech and use them within the same context. The theory we develop is based on concepts from *Service Composition*, *Theorem Proving*, *Information Theory* and *Data*

Emmanouil G. Krasanakis
E-mail: maniospas@hotmail.com

Andreas L. Symmeonidis
E-mail: asymeon@eng.auth.gr

Mining. However, we are not tied down by our approach and also present a set of fully functional tools that can be used to study computer automation.

This work is comprised of the following sections;

Previous Work and Relevant Fields
In this section, we make our first attempts at defining the problem of *Automated Code Generation*. We then examine relevant fields, as well as existing work upon the problem. Much of the presented material has influenced our developed theory. We also defend our decision on using Second-Order logic to approach it.

Introductory Concepts
In this section, we start with well-known concepts of First-Order and Second-Order Logic, but soon develop new definitions and theorems to cover the mathematical needs of our approach. New content builds upon and expands existing work, all the while enriching it with novel ideas. The end-result is a coherent theory that can be used to develop and verify computer automation theories in general. Due to space constraints, trivial proofs have been omitted though.

Automated Code Generation
In this section, we use the defined concepts to formally (i.e. mathematically) state the problem of *Automated Code Generation*. We then proceed to develop theorems which individually contribute towards its solution. Those theorems ensure favorable properties of a proposed Final Algorithm.

Database Import
In this section, we present methods to import data needed for *Automated Code Generation* from already existing libraries; advanced data collection methods further enhance our developed algorithm.

2 Previous Work and Relevant Fields

2.1 Definition for Automated Code Generation

Automated Code Generation can be loosely defined as an automated process (i.e. an algorithm) which, given several statements (i.e. user specifications), derives a suitable implementation in a programming language. To do so, such a process requires an adequate database of implementations and respective user specifications to learn from.

Unfortunately, user specification documents exist only for large projects. And, even then, it may be difficult to match code sections with appropriate specifications. To circumvent these shortcomings, we instead derive user specifications from code comments. Thus, a database for *Automated Code Generation* should contain only well-commented code.

The definition for *Automated Code Generation* we presented, has phenomenal similarities with a number of Relevant Fields. Our work combines methods from those fields, resulting in a focused approach that aims to circumvent their individual shortcomings.

2.2 Relevant Fields

2.2.1 Service Composition

Service Composition is a field where smaller services are combined to form larger ones, according to user specifications. Techniques can be classified as Manual vs Automated or Static vs Dynamic. *Automated Code Generation* bears resemblance to *Static Automated Service Composition* (i.e. composition which is automatically performed during design time and is ready for execution). Relative strategies can be roughly classified in the following categories;[7]

a Workflow-based approaches, which aim towards generating a sequence of actions for solving a given problem according to some user-defined qualities, such as quality-of-service, transactional properties[2] or trust.[12]

b Model-based approaches, which use abstract models -like finite state machines[8]- to adequately represent composition structure. Notable works use dataflow modeling,[9] as well as context awareness and runtime exception handling.[10]

c Artificial Intelligence planning towards generating a composition schema.

d Methods integrating Theorem Proving elements, such as Linear Logic,[3,6] or processing algebraic languages for structuring *Service Composition*.[4,5]

In order to develop methods for *Automated Code Generation*, one could be tempted to split existing programming libraries into small services and apply *Service Composition* methods that maintain logical integrity.[11,12] In this case, appropriate user specification statements could be inferred from relevant code comments. Unfortunately, such an approach suffers from several major drawbacks;

1. Individual services are too strictly-defined to be of practical use in code generation. If services contain a lot of comments or code, they will most likely be too high-level for intricacies to be taken into account or be correctly assimilated into composition logic. Whereas simple services will most likely not reflect the nature of user specifications they adhere to. Instead, it could be far more efficient if composition blurred the borders between different services.

2. There is no guarantee that the composed solution solves the same problem as the one stated. At best, *Service Composition* methods guarantee the non-contradiction between user specifications and the composed solution, but donnot actually *prove* the logical relation between the given problem and the end-result. Therefore, an axiomatic system that allows the conduction of such proofs is desired.

3. Computer code development is more complex than the sequencing of simpler code; code logic usually requires functional composition, such as code nesting. This is impossible for *Service Composition* approaches that use Linear Logic in an attempt to circumvent the previous shortcomings.

After considering the above points, one can see that *Service Composition* closely approaches *Automated Code Generation*, but does not cover the problem adequately, due to inflexible nesting and lack of formalization. Most of all, we lack the appropriate tools to conduct systematic proof-of-concept for each and any method devised.

2.2.2 Theorem Proving

On the other edge of the spectrum, lie the concepts of automated and semi-automated *Theorem Proving*. Relevant methods transform statements within axiomatically well-defined systems (i.e. systems defined through First-order Logic[23]) to either produce simpler statements until a tautological or impossible statement is reached[24–27] or inductively generate new statements.[28,29] In both cases, transformations are indeed both flexible -as first-order logic allows function nesting- and properly formalized. However, those methods also suffer from a couple of drawbacks when applied on *Automated Code Generation*;

1. They cannot produce statements that lie outside the axiomatic system they examine. To circumvent this, we could try to insert axioms that define simplification as the minimization of non-implemented information. In such a case, we still need to properly define the concept of "minimization", as well as the appropriate axioms to do so. The compliance of those axioms with the concept of *Automated Code Generation* must also be proven.

2. It is impossible to take into account less formal systems, such as free text for statements. To do so, we need necessary tools to simultaneously handle free text statements and code expressions.

2.2.3 Information Theory

Automated Code Generation aims to handle information flow. Therefore, it naturally follows that *Information Theory* techniques could be useful to a

certain degree. Generally, *Information Theory* handles the stochastic and statistical analysis of probability distributions, using certain metrics. Those metrics usually aim to measure the uncertainty (i.e. randomness) in a space or stochastic process.

The most well-known of uncertainty metrics is Shannon's Entropy. Entropy has the advantage of being both unique for a given probability distribution of fixed length and consistent,[32] with the uniform distribution giving a local maximum that monotonically increases according to the number of probabilities. In fact, the only functional that satisfies the axioms of continuity, weak additivity, monotonicity, branching and normalization is the Shannon Entropy.[32,33] It has been shown[30] that Shannon Entropy can successfully clasify complex systems in diverse settings that include both deterministic chaotic and stochastic processes.

Concerning the needs for *Automated Code Generation*, we can take advantage of the fact that Shannon'sEntropy can asymptotically approach an optimum estimate for statistical decisions and estimations.[31]

The use of other metrics, such as fuzziness[33] , are also common in information theory. However, as we will handle discrete sets, Shannon's entropy is optimal for any possible needs.

2.2.4 Data Mining

Data Mining tries to discover common patterns throughout available data. Relevant applications can be broadly categorized as;[42]

1. Sequence mining, where patterns are extracted from execution traces or static code traces.
2. Graph mining, where patterns are extracted from call graphs or source code graphs.
3. Text mining, where patterns should be extracted from pure text (such as bug reports, e-mails, code comments and documentation).

Mining methodology may also differ, depending on individual application requirements (i.e. frequent pattern mining, pattern matching, data clustering or data labeling classification).

Being probabilistic in nature, *Data Mining* methods frequently employ Bayesian non-parametric statistics,[43,44] inferring precondition distributions $P(\theta)$ and $P(data|\theta)$ for a finite set of parameters θ and using Bayes's rule to estimate the posterior distribution $P(\theta|data)$. By updating an estimation for θ through this process and then repeating it indefinitely, we can asymptotically estimate the exact values for parameters θ. Bayesian non-parametric statistics have been the subject of intense research in statistics and machine learning,[47] with

popular models including the Dirichlet[45] and Gaussian[46] processes. For graph-based methods, Markov Chains[48,49] are also popular.

The most popular approach is using Tree-Substitution Grammars to perform transformation on Abstract Syntax Trees.[47,48,52] Such methods can be heavily customized and are even able to perform natural language detection.[50–52]

Data Mining can be easily applied on *Automated Code Generation*, by using the correct transformations on Abstract Syntax Trees. Indeed, the notion of flattening for Second-Order Logic is equivalent to Abstract Tree Generation and, thus, transformations we will later introduce for flattened logical models are equivalent to Tree-Substitution Grammars. Unfortunately, the probabilistic analysis is inherently different, as transformation techniques should focus on retaining all original information rather than abstracting it.

2.3 Evaluating Relevant Fields

From the previous analysis, one can see that neither *Service Composition* nor *Theorem Proving* are adequate to tackle the problem of *Automated Code Generation*; trying to develop methods purely within either of those fields would result in crude approximations of the desired functionality. Thus, we must develop hybrid methods that combine the high logical level of *Service Composition* with the logical flexibility of *Theorem Proving*. Also, a formalized theory for exploring such methods should be devised.

A promising approach, which comes really close to *Automated Code Generation* by combining the forementioned fields, is the use of semantics on *Service Composition*.[1] This way, both logical integrity and a degree of flexibility are ensured. However, formalized user input is still needed and results donnot adhere to a hierarchical structure, thus rendering it unsuitable for our needs.

Information Theory provides the fairly useful tool of entropy. However, being stochastic in nature, it lacks the ability to discern possible structural information or handle deterministic rules. Surprisingly though, we will be able to relate our structural metrics with entropy and even achieve similar end-results to maximum likelihood for statistical estimations. Still, the tools we develop provide far better robustness in unfavorable scenarios, while ensuring deterministic optimality in favorable ones.

2.4 Previous Approaches

Automated Code Generation is an emerging field in software engineering and approaches are still both diverse and not universally accepted. This happens due to certain drawbacks of generated code, such as frequent non-intelligibility,

incompleteness (i.e. remainder "TODO" statements) or simply applications that focus on linear problems. Such shortcomings have been pointed out,[14] but approaches have yet to address them effectively; even simple finite-state machine models usually fail during code generation, despite the fact that for such models interpreting is close to detrimental. The primary difficulty lies with the fact that *Automated Code Generation* encompasses problems which differ even on their implementation structure. Thus, an approach that may work well for a problem type may utterly fail on another.

Despite the precarious nature of *Automated Code Generation* though, important milestones have been reached by several approaches. Those can be generally categorized as;[15]

- **Inductive**
  In this case, generated code is inferred from a set of instances. Inductive methods include *Behavioral Cloning* and *Propositional Machine Learning*. Such methods (especially WIM and SYNAPSE for *Propositional Machine Learning*, which is a Turing-Complete process) yield outstanding results in relating concepts and learning simple programs.[15] A specialized inductive approach[53] uses the notions of flattening (i.e. simplifying individual statements of a logical model) and saturation (i.e. adding "background knowledge" to a logical model) to perform transformations on a given First Order Logic model. We used those concepts as a starting point upon which we developed our theoretical framework.

- **Deductive**
  In this case, generated code is inferred from a high-level description. Such methods usually aim to translate the given problem into First-Order Logic, where efficient transformation tools have long been studied.[16] In particular, even older methods are able to perform on approximately the same -and sometimes even higher- level as humans for strictly-defined problems (i.e. Problems stated in First-Order Logic).[17] However, the said translation is not always possible and often undermines effectiveness due to information loss. More recent approaches use evolutionary methods that directly compare implementation and high-level descriptions, thus skipping the hazardous transformation step. Evolutionary methods follow a simple repeatable pattern;[15,18]

  1. Create a random population of individual programs using the functions and terminals available
  2. Run all the programs and compute their fitness
  3. Select (stochastically) the best ones according to some policy
  4. Create a new population by applying random "mutations" on selected individuals (such as inserting variables and function calls, deleting code segments, exchanging code segments or nesting code segments)
  5. Go to 2, until a "good enough" program is found

Of course, the downside to such methods is the computational cost, as the random nature of "mutations" renders convergence towards restrictions both uncertain and slow.

2.5 Approaching Automated Code Generation using Second-Order Logic

In this paper, we develop a fast *deductive* method that works similarly to performing a single "mutation" in each step, with the original "program" being the given description. In order for the "program" to be able to contain both free text and actual code, we use Second-Order Logic, which formulates both human language expressions and programming language statements. Thus, we partially replace parts of the stated problem's description with appropriate implementations, linking inserted implementations. This concept is similar to a variation of evolutionary algorithms called *Probabilistic Incremental Program Evolution*;[15,18]

1. In every generation, the probabilistic model is updated towards the best individual (just one), so that it becomes more likely to generate similar individuals.
2. Minimization (on equal fitnesses, the smaller solution is preferred).

In our case, this method is applied on a single individual that corresponds to the initial problem statement. Also, code insertions move corresponding statements. Our probabilistic model depends on the amount of information being implemented and the amount of information of the generated code, with the assumption that we want less code by removing more statements that result to that code.

For replacing implementations of individual statements, we can use the *Extended Compact Genetic Algorithm* (eCGA), which seems to yield better results that simple *Genetic Programming*.[15] For that algorithm, the following greedy search heuristic is used to find an optimal or near-optimal probabilistic model;[19]

1. Assume each variable is independent of each other. The model is a vector of probabilities.
2. Compute the model complexity and population complexity values of the current model.
3. Consider all possible merges of two variables.
4. Evaluate the model and compressed population complexity values for each model structure.
5. Select the merged model with the lowest combined complexity.

6. If the combined complexity of the best merged model is better than the combined complexity of the model evaluated in step 2, replace it with the best merged model and go to step 2.

7. If the combined complexity of the best merged model is less than or equal to the combined complexity, the model cannot be improved and the model of step 2. is the probabilistic model of the current generation.

In this paper, we modify the above algorithm by changing the way complexity is computed to reflect the structural complexity of expressions according to *Information Theory*,[30] instead of the simpler compressed population complexity based on alphabet cardinality. Also, as mentioned before, instead of minimum combined complexity, we try to minimize a more sophisticated metric that takes into account the amount of information being implemented and the amount of information of the generated code. According to that metric, we also obtain an appropriate greedy rule. The greedy rule we reach, is stochastically optimal, similarly to the maximum likelihood method using entropy in *Information Theory*.[31]

2.6 Formalization of Code Transformations

In order to explore the correctness of concepts discussed, we develop an axiomatic system, upon which appropriate proofs can be conducted. That system bridges the gap between *Theorem Proving* and *Service Composition* methods, allowing the combination and transformation of one field's methods to the other. To do so, we develop tools for handling and comparing Second-Order Logic formulas, as well as ways to interact between overlapping models. Interaction methods expand upon flattening and saturation[53] , by extending those First-Order Logic concepts for Second-Order Logic. Saturation is also extended to the far broader scope of condition application.

Using the developed axiomatic system, we also proceed to actually *proving* the correctness of *Extended Compact Genetic Algorithm* (by showing that it converges on a global minimum for non-intersecting sets of comments and their implementation code). We also modify the greedy rule so that it yields linear execution time for each step, while still obtaining the required results for the average case - much like maximum likelihood in *Information Theory*, as stated previously.

Essentially, we donnot limit ourselves to just using the forementioned *Probabilistic Inrcemental Program Evolution* and *Extended Compact Genetic Algorithm*, but we also prove their effectiveness and conditional optimality in the average case (proof-of-concept).

3 Introductory Concepts

In this section, we develop theoretical tools for handling Second-Order Logic models. In order to make reading more comprehensible, we present only the most necessary non-trivial and non-intermediary concepts. For full understanding, as well as complete coverage of concepts silently ignored here, missing material can be found in the complete -but more obscure- version of this work.

Table 1 All Introductory Definitions

Functions
Predicates
Terms
Formulas
Implication
Free Variables
Quantified Variables
q-Complexity
Multiplicity Union
Two-Way Implcation
Equivalue
Equation
Formula Words
Lexicographic Similarity
Model
Multiplicity Model Union
Model Subtraction
Equal Models
Equivalent Models
Formula Transformation
Model Formula Properties
Model q-Complexity
Model Size
Equivalue Subtraction
Equivalue Subtraction upon Multiplicity Union
Connected Model
Independent Models
Complete Model
Sub-Model Set

Table 2 All Introductory Theorems

Formulas yield logical values
1-Complexity and Entropy
Independent Quantifiers
Equality Implies Equivalence
Formula Transformation Retains Equality
Equivalence to Equivalue
Equality to Two-Way Implication
Multiplicity Union Properties
q-Complexity of Formula Transformation
Model Name
Connectivity of Equivalent Models
Subsets of Independent Models
Connected Models are Complete
Independent Complete Models
Minimum q-Complexity for Formulas in Complete Models
Direct Algebrization
Inverse Algebrization
Condition Application
Flattening
Flattening Retains Completeness and Connectivity
Flattening and Entropy
Sub-Model Construction
Non-Referenced Existential Quantifier
Non-Referenced Iterative Quantifier

Definition 1 (q-Complexity) Let $\|\{v_1, ..., v_n\}\|_q = (v_1^q + ... + v_n^q)^{\frac{1}{q}}$, $v_i \in \mathbb{R}$ denote the q-norm for real numbers.
We define recursively the q-complexity $\|f\|_q$ as;

- $\|f\|_q = 0 \Leftrightarrow f = \emptyset$
- $\|f\|_q = 1 \Leftrightarrow f \in \{variables\} \cup \{constants\}$
- $\|P(t_1, ..., t_n)\|_q = \big\|\{\|t_1\|_q, ..., \|t_n\|_q\}\big\|_q + 1$, $P \in \{functions\} \cup \{predicates\}$
- $\|a = b\|_q = \big\|\{\|a\|, \|b\|\}\big\|_q + 1$
- $\|a \circ b\|_q = \big\|\{\|a\|, \|b\|\}\big\|_q + 1$ for $a, b \neq \emptyset$ and $\circ \in \{\leftrightarrow, \rightarrow, and, or\}$
- $\|^\circ x\, f\|_q = \|\{1, \|f\|_q\}\|_q + 1$ where x is a variable and $^\circ \in \{\exists, \forall\}$

When not stated otherwise, we use the term complexity for the 1-complexity $\|f\|_1 = \|f\|$, where the norm is reduced to simple addition. We also use the term depth for $\|f\|_\infty = |f|$, where the norm yields the max function.

Contrary to classical definition, where the depth of a single variable is considered 0, we will work with depth of variables being 1 (so the classical definition could be obtained for $|f| \leftarrow |f| - 1$). This way, we ensure that the equation $\|f\|_q = 0$ has a unique solution.

Definition 2 (Multiplicity Union) The multiplicity union $\oplus$ between set $\mathcal{A}$ and $\mathcal{B}$ such that $\emptyset \notin \mathcal{A} = \{a_1, ..., a_n\}$ and $\emptyset \notin \mathcal{B} = \{b_1, ..., b_m\}$ is defined as $\mathcal{A} \oplus \mathcal{B} = \{a_1, ..., a_n, b_1, ..., b_m\}$ (arraying elements of $\mathcal{B}$ after elements of $\mathcal{A}$). We will extend this definition later for taking into account sets that contain empty sets or empty formulas.

In order to compare formulas from a programmer's standpoint, we also define the following relation;

Definition 3 (Equivalue) We abstractly define the equivalue relation $\sim$ that adheres to the following restrictions $\forall a, b \in \mathbb{SOL}$;

$$\mathcal{R}[a\,and\,b] \sim \mathcal{R}[a] \cup \mathcal{R}[b], \mathcal{R} = \mathcal{V}, \mathcal{X}, \mathcal{V} - \mathcal{X} \tag{1a}$$

$$\big(\mathcal{A} \sim \mathcal{C}\,and\,\mathcal{B} \sim \mathcal{D}\big) \Rightarrow \big(\mathcal{A} \cup \mathcal{B} \sim \mathcal{C} \cup \mathcal{D}\,and\,\mathcal{A} \oplus \mathcal{B} \sim \mathcal{C} \oplus \mathcal{D}\big) \tag{1b}$$

$$\big(a \leftrightarrow b\,and\,\mathcal{V}[a] \sim \mathcal{V}[b]\,and\,(\mathcal{V} - \mathcal{X})[a] \sim (\mathcal{V} - \mathcal{X})[b]\big) \Rightarrow a \sim b \tag{1c}$$

$$a \sim b \Leftrightarrow \Big[\big(\mathcal{V}[a] - \mathcal{X}[a] \sim \mathcal{V}[b] - \mathcal{X}[b]\big) \Rightarrow \mathcal{X}[a] \sim \mathcal{X}[b]\Big] \tag{1d}$$

In practice, the stated restrictions respectively translate to;

- the ability to run blocks of code independently (e.g. sequentially)
- the ability to replace variables
- the fact that two blocks of code with the same variables that are logically equivalent have the same behavior
- the definition that two blocks of code are equivalue if and only if they have the same free variable values for the same quantifier values

The notion of equivalue allows us to handle formulas as "black boxes", whose internal behavior (i.e. free variables) is the same for the same observable behavior (i.e. quantified variables - we later see that quantifiers correspond to inputs and outputs).

Definition 4 (Lexicographic Similarity) For two formulas $M, N \in \mathbb{SOL}$, we define lexicographic similarity as;

$$lex(M, N) = \frac{|\mathcal{W}[M] - \mathcal{V}[M] - \mathcal{W}[N]| + |\mathcal{W}[N] - \mathcal{V}[N] - \mathcal{W}[M]|}{|\mathcal{W}[M] - \mathcal{V}[M]| + |\mathcal{W}[N] - \mathcal{V}[N]|} \tag{2}$$

.

3.0.1 Models and Properties

Definition 5 (Model) A model $\mathcal{M} \in \mathbb{SOL}^*$ is a non-ordered finite set of formulas $\mathcal{M}_i \in \mathbb{SOL}$. The empty set ($\emptyset$) is also a model.

Definition 6 (Multiplicity Model Union) For models $\mathcal{M}, \mathcal{N}$ comprised of formulas $\mathcal{M}_i \in \mathbb{SOL}$ and $\mathcal{N}_j \in \mathbb{SOL}$ respectively, we define their multiplicity union as;

$$\mathcal{M} \oplus \mathcal{N} = \{\mathcal{M}_i : \mathcal{M}_i \neq \emptyset\} \oplus \{\mathcal{N}_j : \mathcal{N}_j \neq \emptyset\} \tag{3}$$

3.0.2 Lindenbaum-Tarski Algebrization and Equivalue for Models

Definition 7 (Formula Transformation) We define the $1-1$ correspondence $formula : \mathbb{SOL}^* \to \mathbb{SOL}$ as;

$$formula(\mathcal{M}_1 \oplus ... \oplus \mathcal{M}_m) = \mathcal{M}_1 \, and \, ... \, and \, \mathcal{M}_m \tag{4}$$

for $M_i \in \mathbb{SOL}$ and $\mathcal{M}_i \neq \emptyset$.

Theorem 1 (Formula Transformation Retains Equality) *For models $\mathcal{M}, \mathcal{N} \in \mathbb{SOL}^*$, the formula transformation has the property;*

$$\mathcal{M} = \mathcal{N} \Leftrightarrow formula(\mathcal{M}) = formula(\mathcal{N}) \tag{5}$$

Definition 8 (Model Formula Properties) For models $\mathcal{M}, \mathcal{N} \in \mathbb{SOL}^*$ we define;

$$\mathcal{M} \sim \mathcal{N} \Leftrightarrow formula(\mathcal{M}) \sim formula(\mathcal{N}) \tag{6a}$$
$$\mathcal{M} \leftrightarrow \mathcal{N} \Leftrightarrow formula(\mathcal{M}) \leftrightarrow formula(\mathcal{N}) \tag{6b}$$
$$\mathcal{V}[\mathcal{M}] = \mathcal{V}[formula(\mathcal{M})] \tag{6c}$$
$$\mathcal{X}[\mathcal{M}] = \mathcal{X}[formula(\mathcal{N})] \tag{6d}$$

Theorem 2 (Multiplicity Union Properties) *For models $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D} \in \mathbb{SOL}^*$ the following hold true;*

$$\mathcal{A} \oplus \mathcal{B} = \mathcal{B} \oplus \mathcal{A} \tag{7a}$$
$$\big(\mathcal{A} \leftrightarrow \mathcal{C} \, and \, \mathcal{B} \leftrightarrow \mathcal{D}\big) \Rightarrow \mathcal{A} \oplus \mathcal{B} \leftrightarrow \mathcal{C} \oplus \mathcal{D} \tag{7b}$$
$$\big(\mathcal{A} \sim \mathcal{C} \, and \, \mathcal{B} \sim \mathcal{D}\big) \Rightarrow \mathcal{A} \oplus \mathcal{B} \sim \mathcal{C} \oplus \mathcal{D} \tag{7c}$$

Proof

– Formulas in $\mathcal{A} \oplus \mathcal{B}$ will have the same multiplicity with formulas in $\mathcal{B} \oplus \mathcal{A}$.Thus, $\mathcal{A} \oplus \mathcal{B} = \mathcal{B} \oplus \mathcal{A}$.

- $\mathcal{A} \leftrightarrow \mathcal{C} \Leftrightarrow formula(\mathcal{A}) \leftrightarrow formula(\mathcal{C})$.
  Similarly, $formula(\mathcal{B}) \leftrightarrow formula(\mathcal{D})$.
  Thus, $formula(\mathcal{A})\, and\, formula(\mathcal{B}) \leftrightarrow formula(\mathcal{C})\, and\, formula(\mathcal{D})$
  $\Leftrightarrow formula(\mathcal{A} \oplus \mathcal{B}) \leftrightarrow formula(\mathcal{C} \oplus \mathcal{D}) \Leftrightarrow$
  $\mathcal{A} \oplus \mathcal{B} \leftrightarrow \mathcal{C} \oplus \mathcal{D}$.
- $\mathcal{A} \sim \mathcal{C} \Leftrightarrow formula(\mathcal{A}) \sim formula(\mathcal{C})$.
  Therefore, $\mathcal{R}[formula(\mathcal{A})] \sim \mathcal{R}[formula(\mathcal{C})]$ for $\mathcal{R} = \mathcal{X}, \mathcal{V}, \mathcal{V} - \mathcal{X}$.
  Similarly, $\mathcal{R}[formula(\mathcal{B})] \sim \mathcal{R}[formula(\mathcal{D})]$.
  Thus, from equivalue properties,
  $\mathcal{R}[formula(\mathcal{A})] \cup \mathcal{R}[formula(\mathcal{B})] \sim \mathcal{R}[formula(\mathcal{C})] \cup \mathcal{R}[formula(\mathcal{D})]$.
  Now, from equivalue properties, it also follows that
  $\mathcal{R}[formula(\mathcal{A})\, and\, formula(\mathcal{B})] \sim \mathcal{R}[formula(\mathcal{A})] \cup \mathcal{R}[formula(\mathcal{B})]$
  as well as
  $\mathcal{R}[formula(\mathcal{C})\, and\, formula(\mathcal{D})] \sim \mathcal{R}[formula(\mathcal{C})] \cup \mathcal{R}[formula(\mathcal{D})]$.
  Consequently,
  $\mathcal{R}[formula(\mathcal{A})\, and\, formula(\mathcal{B})] \sim \mathcal{R}[formula(\mathcal{C})\, and\, formula(\mathcal{D})]$.
  Now, using equivalue properties again, we finally obtain
  $formula(\mathcal{A})\, and\, formula(\mathcal{B}) \sim formula(\mathcal{C})\, and\, formula(\mathcal{D}) \Leftrightarrow$
  $formula(\mathcal{A} \oplus \mathcal{B}) \sim formula(\mathcal{C} \oplus \mathcal{D}) \Leftrightarrow \mathcal{A} \oplus \mathcal{B} \sim \mathcal{C} \oplus \mathcal{D}$.

Definition 9 (Model q-Complexity) For model $\mathcal{M} = \mathcal{M}_1 \oplus ... \oplus \mathcal{M}_m$ and q-Norm $\|\cdot\|_q$ we define q-Compxity for models as;

$$\|\mathcal{M}\|_q = \big\|\{\|\mathcal{M}_1\|_q, ..., \|\mathcal{M}_m\|_q\}\big\|_q \tag{8}$$

Again, when not otherwise stated, complexity will refer to $\|\mathcal{M}\| = \|\mathcal{M}\|_1$ and we define a model's depth as $|\mathcal{M}| = \|\mathcal{M}\|_\infty$.

Definition 10 (Equivalue Subtraction) The equivalue subtraction between two models $\mathcal{M}, \mathcal{N} \in \mathbb{SOL}^*$ is defined as;

$$\mathcal{M} \ominus \mathcal{N} = \mathcal{M} - \oplus_{\mathcal{K} \subseteq \mathcal{M}, \mathcal{K} \sim \mathcal{N}} \mathcal{K} \tag{9}$$

Theorem 3 (Equivalue Subtraction upon Multiplicity Union) *For models $\mathcal{M}, \mathcal{N} \in \mathbb{SOL}^*$, it holds that;*

$$(\mathcal{M} \oplus \mathcal{N}) \ominus \mathcal{N} = \mathcal{M} \ominus \mathcal{N} \tag{10}$$

3.0.3 Algebrization

Theorem 4 (Direct Algebrization) *There exists a function connector* : $\mathbb{SOL}^* \times \mathbb{SOL}^* \to \mathbb{SOL}^*$ *such that, for any two independent models* $\mathcal{M}, \mathcal{N} \in \mathbb{SOL}^*$*;*

$$\mathcal{M} \sim \mathcal{N} \Rightarrow \Big[\exists \mathcal{C} = connector(\mathcal{M}, \mathcal{N}) \quad \mathcal{M} \oplus \mathcal{C} \leftrightarrow \mathcal{N} \quad \mathcal{N} \oplus \mathcal{C} \leftrightarrow \mathcal{M}\Big] \tag{11}$$

Proof
For formulas $M\ N$, we can use the model $\mathcal{C} = connector(M,N) = \{\mathcal{V}[M] \sim \mathcal{V}[N]\}$. Hence, we can extend this operation to models as $connector(\mathcal{M},\mathcal{N}) = connector\big(formula(\mathcal{M}), formula(\mathcal{N})\big)$.

Theorem 5 (Inverse Algebrization) *There exists a function equator* : $\mathbb{SOL}^* \times \mathbb{SOL}^* \to \mathbb{SOL}^*$ *such that, for any two models* $\mathcal{M},\mathcal{N} \in \mathbb{SOL}^*$*;*

$$\mathcal{M} \leftrightarrow \mathcal{N} \Rightarrow \Big[\exists\, \mathcal{Q} = equator(\mathcal{M},\mathcal{N}) \quad \mathcal{M} \oplus \mathcal{Q} \sim \mathcal{N} \oplus \mathcal{Q}\Big] \tag{12}$$

Proof

Let us denote $x_M = \mathcal{X}[\mathcal{M}], x_N = \mathcal{X}[\mathcal{N}], t_M = \mathcal{V}[\mathcal{M}] - \mathcal{X}[\mathcal{M}], t_n = \mathcal{V}[\mathcal{N}] - \mathcal{X}[\mathcal{N}]$.
We can construct the required model as $\mathcal{Q} = \exists(x_M \cup x_N - x_M \cap x_n)\,\emptyset \oplus \forall(t_M \cup t_N - t_M \cap t_N)\,\emptyset$.

For this model, $\mathcal{M} \leftrightarrow \mathcal{N} \Rightarrow \mathcal{Q} \oplus \mathcal{M} \leftrightarrow \mathcal{Q} \oplus \mathcal{N} \Leftrightarrow formula(\mathcal{Q} \oplus \mathcal{M}) \leftrightarrow formula(\mathcal{Q} \oplus \mathcal{N})$
Additionally, $\mathcal{V}\big[formula(\mathcal{Q} \oplus \mathcal{M})\big] = x_{\mathcal{M}} \cup t_{\mathcal{M}} \cup x_{\mathcal{N}} \cup t_{\mathcal{N}} = \mathcal{V}\big[formula(\mathcal{Q} \oplus \mathcal{N})\big] \Rightarrow$
$\boxed{\mathcal{V}\big[formula(\mathcal{Q} \oplus \mathcal{M})\big] \sim \mathcal{V}\big[formula(\mathcal{Q} \oplus \mathcal{N})\big]}$
Also, $\mathcal{X}\big[formula(\mathcal{Q} \oplus \mathcal{M})\big] = x_{\mathcal{M}} \cap x_{\mathcal{N}} = \mathcal{X}\big[formula(\mathcal{Q} \oplus \mathcal{N})\big] \Rightarrow$
$\boxed{\mathcal{X}\big[formula(\mathcal{Q} \oplus \mathcal{M})\big] \sim \mathcal{X}\big[formula(\mathcal{Q} \oplus \mathcal{N})\big]}$
Therefore, due to equivalue properties, we obtain the assertion $formula(\mathcal{Q} \oplus \mathcal{M}) \sim formula(\mathcal{Q} \oplus \mathcal{N}) \Leftrightarrow \mathcal{Q} \oplus \mathcal{M} \sim \mathcal{Q} \oplus \mathcal{N}$.

Theorem 6 (Condition Application) *For any two (not necessarily independent) models* $\mathcal{M},\mathcal{N} \in \mathbb{SOL}^*$ *we can define;*

$$connector(\mathcal{M},\mathcal{N}) = \oplus_{\mathcal{K}\subseteq\mathcal{M},\mathcal{K}\sim\mathcal{N}} connector(\mathcal{K},\mathcal{N}) \tag{13}$$

for which it holds that;

$$(\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{N} \oplus connectoor(\mathcal{M},\mathcal{N}) \leftrightarrow \mathcal{M} \tag{14}$$

Proof

From the theorem subsets of independent models we obtain that for $\mathcal{K} \subseteq \mathcal{M}$ the models $\mathcal{K},\mathcal{N}$ are independent. Therefore, for $\mathcal{K} \sim \mathcal{N}$, we can use direct algebrization and obtain the model $connector(\mathcal{K},\mathcal{N})$.

Let us now select $\mathcal{L} \in \mathbb{SOL}^*$ such that;

$$(\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{L} = \mathcal{M}$$

Obviously, the following also holds true;

$$\mathcal{N} \ominus \big((\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{L}\big) = \mathcal{N} \ominus \mathcal{L}$$

Hence, $\forall \mathcal{K} \sim \mathcal{N}$ we obtain that;

$$\Big[\mathcal{K} \sim \big((\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{L}\big)\Big] \Leftrightarrow \mathcal{K} \sim \mathcal{L}$$

Thus;

$$\begin{aligned} connector(\mathcal{M}, \mathcal{N}) &= connector\big((\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{L}, \mathcal{N}\big) \\ &= \oplus_{\mathcal{K} \subset (\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{L}, \mathcal{K} \sim \mathcal{N}} connector(\mathcal{K}, \mathcal{N}) \\ &= \oplus_{\mathcal{K} \subset \mathcal{L}, \mathcal{K} \sim \mathcal{N}} connector(\mathcal{K}, \mathcal{N}) \\ &= connector(\mathcal{L}, \mathcal{N}) \end{aligned}$$

Additionally, for $\mathcal{L} \sim \mathcal{K}$, it most hold true that;

$$\begin{gathered} \mathcal{L} = \mathcal{L}_1 \oplus ... \oplus \mathcal{L}_n \\ \mathcal{L}_i \sim \mathcal{K} \, \forall i \end{gathered}$$

Therefore, we can construct a two-way implication;

$$\begin{aligned} \oplus_i \mathcal{L}_i &\leftrightarrow \oplus_i \big(\mathcal{N} \oplus connector(\mathcal{L}_i, \mathcal{N})\big) \\ &= (\oplus_i \mathcal{N}) \oplus \big(\oplus_i connector(\mathcal{L}_i, \mathcal{N})\big) \\ &\leftrightarrow \mathcal{N} \oplus \big(\oplus_i connector(\mathcal{L}_i, \mathcal{N})\big) \\ &= \mathcal{N} \oplus connector(\mathcal{L}, \mathcal{N}) \end{aligned}$$

Finally, as $(\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{L} = \mathcal{M}$, we derive the desired result;

$$(\mathcal{M} \ominus \mathcal{N}) \oplus \mathcal{N} \oplus connector(\mathcal{M}, \mathcal{N}) \leftrightarrow \mathcal{M}$$

This theorem is the expansion of the traditional concept of saturation.[53] The $\ominus$ operation does excellent work in generalizing our model while maintaining two-way implication (this effect is known in the literature as "dropping conditions"[53,54]).

Theorem 7 (Flattening) *There exists a function* $flattened : \mathbb{SOL}^* \to \mathbb{SOL}^*$ *such that for any model* $\mathcal{M} \in \mathbb{SOL}^*$*;*

$$\mathcal{M} \leftrightarrow flattened(\mathcal{M}) \tag{15a}$$

$$|flattened(\mathcal{M})| \leq 2 \tag{15b}$$

Proof

If M contains only variables, constants and empty expressions, we define $formula(\mathcal{M}) = \mathcal{M}$, thus $|formula(\mathcal{M})| = |\mathcal{M}| \leq 1$. Let us also assume that for models with depth $\leq n$ we can define their flattened model (strong induction). We proceed to show that, as a result, the same must hold true for $|\mathcal{M}| = n + 1$.

To prove the above, it suffices to show it for formulas M with the same depth $n + 1$, as for any model $\mathcal{M} = M_1 \oplus ... \oplus M_n$ with $M_i \in \mathbb{SOL}$ we obtain

$|\mathcal{M}| \leq n+1 \Leftrightarrow |M_i| \leq n+1$ and, thusly, we can generate $formula(\mathcal{M}) = \oplus_i formula(M_i)$. To recap; it suffices to show that using $flattened(\mathcal{M})$ for models $|\mathcal{M}| \leq n$, we can generate $flattened(M)$ for $|M| = n+1$. To show that this is possible, we examine all possible cases;

- If $M = a \circ b$ with $a, b \neq \emptyset$ then $|a| \leq n, |b| \leq n$ and, as a result, $|flattened(a)| \leq 2, |flattened(b)| \leq 2$. Therefore, $flattened(M) = (\exists c\, \emptyset) \oplus (c = flattened(a)) \oplus (\exists d\, \emptyset) \oplus (d = flattened(b)) \oplus (c \circ d)$.
  We assert that $|flattened(M)| = \max\{|\exists c\, \emptyset|, |c = flattened(a)|, |\exists d\, \emptyset|, |d = flattened(b)|, |c \circ d|\} = \max\{\leq 2, \leq 2, 1, 1\} \leq 2$.
- If $M =^{\circ} a\, b$ (a is a variable) then $|b| = n$ and, as a result, $|flattened(b)| \leq 2$. Therefore, $flattened(M) = (\exists c\, \emptyset) \oplus (c = flattened(b)) \oplus (\exists a\, \emptyset)$.
  We assert that $|flattened(M)| = \max\{|\exists c\, \emptyset|, |c = flattened(b)|, |\exists a\, \emptyset|\} = \max\{\leq 2, 1\} = 2$.
- If $M = P(t_1, ..., t_n)$ (t_1 are terms) then $|t_i| \leq n$ and, thus, $|flattened(t_i)| \leq 2$. We can now define $flattened(M) = \oplus_i \big((\exists x_i\, \emptyset) \oplus (x_i = flattened(t_i))\big) \oplus (P(x_1, ..., x_n))$.
  We assert that $|flattened(M)| = \max\{|(P(x_1, ..., x_n)|, \max_i\{|\exists x_i\, \emptyset|, |x_i = flattened(t_i)|\}\} = \max\{2, \max\{1, \leq 2\}\} = 2$.
- If $M = (a = b)$ then $|a| \leq n, |b| \leq n$. If, additionally, $|flattened(a)| \leq 2, |flattened(b)| \leq 2$, it follows that $flattened(M) = (flattened(a) = flattened(b))$.
  We assert that $|flattened(M)| = \max\{|flattened(a)|, |flattened(b)|\} = \max\{\leq 2, \leq 2\} \leq 2$.
- If $M = (a = b)$ with $|flattened(a)| > 2$ (the case not covered by the above point), then a must necessarily also be of the form $a = (a_1 = a_2)$. However, following the same reasoning, either a_1 or a_2 must necessarily be an equation between two terms. Therefore, we must infinitely descend in this case, reaching the conclusion that $n + 1 = |M| = \infty$, which is impossible.

This theorem expands the traditional flattening method[53] to work for Second-Order Logic. The two-way implication is detrimental, thanks to the construction process.

3.0.4 Sub-Models

Sub-models is a formalization in second-order logic for sub-problems,[22] which are discrete parts that contribute to the solution of a bigger problem. We can take advantage of developed theory to present the following definition;

Definition 11 (Sub-Model Set) For model $\mathcal{M} \in \mathbb{SOL}^*$ we define its set of sub-models $models[\mathcal{M}] \subseteq \mathcal{M}^*$ as the minimum set such that $\oplus models[\mathcal{M}] \equiv \mathcal{M}$, $\mathcal{M}_t$ is connected $\forall \mathcal{M}_t \in models[\mathcal{M}]$ and $\sum_{\mathcal{M}_t \in models[\mathcal{M}]} \|\mathcal{M}_t\| = \max$.

Theorem 8 (Sub-Model Construction) *To construct the sub-model set* $models[\mathcal{M}]$ *of a model* $\mathcal{M} \in \mathbb{SOL}^*$*, we generate the graph* $\mathcal{G}_{\mathcal{M}} = \big(\mathcal{V}[\mathcal{M}], \mathcal{E}\big)$ *with edges* $\mathcal{E} \subseteq \mathcal{V}[\mathcal{M}] \times \mathcal{V}[\mathcal{M}]$ *such that* $(v_1, v_2) \in \mathcal{E} \Leftrightarrow [\exists M \in \mathcal{M} \subseteq \mathbb{SOL} : v_1 \in M, v_2 \in M]$.
We then define the set $\mathcal{G}^*_{\mathcal{M}}$ *as the maximum connected subgraphs of* $\mathcal{G}_{\mathcal{M}}$ *and define the function* $f_{\mathcal{M}}(\mathcal{S}) = \oplus\{M : M \in \mathcal{M}, \exists v \in M : v \in \mathcal{V}[\mathcal{S}]\}$*, where* $\mathcal{V}[\mathcal{S}]$ *the vertexes of the graph* $\mathcal{S}$.
Now, the sub-model set can be constructed as

$$models[\mathcal{M}] = \Big\{ f_{\mathcal{M}}(\mathcal{S}), \mathcal{S} \in \big(\mathcal{G}_m - (\mathcal{V} - \mathcal{X})[\mathcal{M}]\big)^* \Big\} \tag{16}$$

4 Automated Code Generation

4.1 Defining the Problem

Definition 12 (Automated Code Generation) Let us assume sets of models $\mathbb{D}, \mathbb{I} \subseteq \mathbb{SOL}^*$ and function $t : \mathbb{SOL}^* \to \mathbb{SOL}^*$ for which $t(\mathcal{M}) \sim \mathcal{M} \forall \mathcal{M} \in \mathbb{SOL}^*$, $t(\mathbb{D}) = \mathbb{I}$ and $connector\big(\mathcal{M}, t(\mathcal{M})\big)$ is known $\forall \mathcal{M} \in \mathbb{D}$. We need to define the function t outside $\mathbb{D}$, so that, for any given model $\mathcal{H} \in \mathbb{SOL}^*$, we obtain $t(\mathcal{H}) \ominus (\oplus \mathbb{I}) \ominus \{y = x, \forall y\, \emptyset, \exists x\, \emptyset\} = \emptyset$ (where x, y arbitrary values).

4.2 Sets of Models

4.2.1 Independence on Sets of Models

Theorem 9 (Independence within Sets of Models) *For any finite set of finite (i.e. with finite complexity) models* $\mathbb{S} \subseteq \mathbb{SOL}^*$ *exists a finite set of finite models* $\mathbb{S}' \subseteq \mathbb{SOL}^*$ *such that;*

$$\forall \mathcal{M} \in \mathbb{S}\, \exists \mathcal{M}' \in \mathbb{S}' \quad \mathcal{M} \sim \mathcal{M}' \tag{17a}$$
$$\forall \mathcal{M}', \mathcal{N}' \in \mathbb{S}' \quad \mathcal{M}', \mathcal{N}' \, are \, independent \tag{17b}$$
$$|\mathbb{S}'| = |\mathbb{S}| \tag{17c}$$
$$\| \oplus \mathbb{S}' \| = \| \oplus \mathbb{S} \| \tag{17d}$$

(i.e. It generates a new set of independent models which are equivalue to those of the previous set, retaining both the number of models and total complexity.)
We define a process for generating such a transformation $independent : 2^{\mathbb{SOL}^*} \to 2^{\mathbb{SOL}^*}$ *(i.e.* $\mathbb{S}' = independent(\mathbb{S})$*).*

Theorem 10 (Independence from a Set of Models) *For a finite set of finite independent models $\mathbb{S} \subseteq \mathbb{SOL}^*$ exists a transformation independent$_{\mathbb{S}}$: $\mathbb{SOL}^* \to \mathbb{SOL}^*$ such that for any finite model $\mathcal{H} \in \mathbb{SOL}^*$;*

$$\mathcal{H} \sim independent_{\mathbb{S}}(\mathcal{H}) \tag{18a}$$

$$\forall \mathcal{S} \in \mathbb{S} \quad independent_{\mathbb{S}}(\mathcal{H}), \mathcal{S} \, are \, independent \tag{18b}$$

$$\|\mathcal{H}\| = \|independent_{\mathbb{S}}(\mathcal{H})\| \tag{18c}$$

Theorem 11 (Independence Retaining Equivalue) *For sets of models $\mathbb{D}, \mathbb{I} \subseteq \mathbb{SOL}^*$ and function $t_g : \mathbb{SOL}^* \to \mathbb{SOL}^*$ for which $t_g(\mathcal{M}) \sim \mathcal{M} \forall \mathcal{M} \in \mathbb{SOL}^*$ and $t_g(\mathbb{D}) = \mathbb{I}$ exist sets of models $\mathbb{D}_g, \mathbb{I}_g \subseteq \mathbb{SOL}^*$ for which $t_g(\mathbb{D}_g) = \mathbb{I}_g$ and the properties presented in independence within sets of models hold true for both the pair $\mathbb{D}, \mathbb{D}_g$ and the pair $\mathbb{I}, \mathbb{I}_g$.*

4.2.2 Equivalueless q-Complexity and Information Purity

Definition 13 (Equivalueless q-Complexity between Sets of Models) For sets of models $\mathbb{D}, \mathbb{F} \subseteq \mathbb{SOL}^*$ with $\mathbb{D} \cup \mathbb{F}$ being ordered, we define the equivalueless complexity of $\mathbb{F}$ compared to $\mathbb{D}$ as $\|\mathbb{F}\|_{q,\mathbb{D}} = \left\| \oplus_{\mathcal{F} \in \mathbb{F}} \left(\mathcal{F} \ominus (\oplus_{\mathcal{D} \in \mathbb{D}, \mathcal{F} < \mathcal{D}} \mathcal{D}) \right) \right\|_q$. When not stated otherwise, we use the equivalueless 1-complexity $\|\mathbb{F}\|_{\mathbb{D}} = \|\mathbb{F}\|_{1,\mathbb{D}} = \sum_{\mathcal{F} \in \mathbb{F}} \left\| \mathcal{F} \ominus (\oplus_{\mathcal{D} \in \mathbb{D}, \mathcal{F} < \mathcal{D}} \mathcal{D}) \right\|$.

Theorem 12 (Lower Bound for Equivalueless q-Complexity) *For sets of models $\mathbb{D}, \mathbb{F} \subseteq \mathbb{SOL}^*$ with $\mathbb{D} \cup \mathbb{F}$ being ordered,*

$$\|\mathbb{F}\|_{\mathbb{D}} \geq \left\| (\oplus \mathbb{F}) \ominus (\oplus \mathbb{D}) \right\| \tag{19}$$

with the equality holding if and only if $\mathcal{F} < \mathcal{D} \, \forall \mathcal{F} \in \mathbb{F}, \mathcal{D} \in \mathbb{D}$.

Definition 14 (Equivalueless q-Complexity for a Set of Models) For an ordered set of models $\mathbb{D}$, we define its equivalueless q-complexity as;

$$\|\mathbb{D}\|_q = \|\mathbb{D}\|_{q,\mathbb{D}} \tag{20}$$

Definition 15 (Relative Information Purity) For sets of models $\mathbb{D}, \mathbb{F} \subseteq \mathbb{SOL}^*$ with $\mathbb{D} \cup \mathbb{F}$ being ordered, we define relative information purity as;

$$p_{\mathbb{D}}(\mathbb{F}) = \min_{\mathbb{O} \in 2^{\mathbb{F}}} \frac{\|\mathbb{O}\|_{\mathbb{D}}}{\|\mathbb{F}\|_{\emptyset}} \tag{21}$$

4.3 Model Transformations

Definition 16 (Detrimental Model) Using arbitrary variables x, y. we define the detrimental model as;

$$\mathbb{Z} = \{y = x, \forall y\, \emptyset, \exists x\, \emptyset\} \tag{22}$$

Due to the way the *connector* and *equator* functions were defined in their relevant proofs, we obtain that they are equivalue to the detrimental model.

4.3.1 Simplifying the Problem

The definition for automated code generation is too strict to comfortably apply Algebrization. Thus, we define the problem in a more loose manner;

Definition 17 (Loose Automated Code Generation) Let us assume sets of models $\mathbb{D}_g, \mathbb{I}_g \subseteq \mathbb{SOL}^*$ and function $t_g : \mathbb{SOL}^* \to \mathbb{SOL}^*$ for which;

- $t_g(\mathcal{D}) \sim \mathcal{D}\, \forall \mathcal{D} \in \mathbb{D}$
- $t_g(\mathbb{D}) = \mathbb{I}$
- $connector\big(\mathcal{D}, t_g(\mathcal{D})\big)$ is known $\forall \mathcal{D} \in \mathbb{D}$

We need to define the function t_g outside $\mathbb{D}$, so that for a given model $\mathcal{H} \in \mathbb{SOL}^*$ exists a model $\mathcal{P} \in \mathbb{SOL}^*$ such that $(\mathcal{P} - \mathcal{H}) \ominus \mathbb{Z} = \emptyset$, $t_g(\mathcal{H}) \sim \mathcal{P}$ and $t_g(\mathcal{H}) \ominus (\oplus\mathbb{I}) \ominus \mathbb{Z} = \emptyset$.

To solve the above problem, we define a sequence of model transformations $\{\mathcal{H}_{n+1}\}_{n\in\mathbb{N}}$, which correspond to *Extended Compact Genetic Algorithm* mutations.[19]

Definition 18 (General Algorithm for Automated Code Generation) For sets of models $\mathbb{D}_g, \mathbb{I}_g$ and function t_g defined for loose automated code generation, starting model $\mathcal{H} \in \mathbb{SOL}^*$ and sequence of models $\{\mathcal{D}_n\}_{n\in\mathbb{N}}, \mathcal{D}_n \in \mathbb{D}_g$ we define $\{\mathcal{H}_n\}_{n\in\mathbb{N}}$ according to the following algorithm;

$$\mathcal{C}_n = connector\big(\mathcal{H}_n, t_g(\mathcal{D}_n)\big) \oplus equator(\mathcal{D}_n, \mathcal{H}_n) \tag{23a}$$

$$\mathcal{H}_{n+1} = independent_{\mathbb{D}_g}\Big((\mathcal{H}_n \ominus \mathcal{D}_n) \oplus t_g(\mathcal{D}_n) \oplus \mathcal{C}_n\Big) \tag{23b}$$

$$\mathcal{H}_0 = \mathcal{H} \tag{23c}$$

where $connector\big(\mathcal{H}_n, t_g(\mathcal{D}_n)\big) = connector\big(\mathcal{H}_n, t_g(\mathcal{D}_n)\big) \oplus connector\big(\mathcal{D}_n, t_g(\mathcal{D}_n)\big)$ and $\mathcal{H}_n \ominus \mathcal{D}_n \neq \mathcal{H}_n\, \forall n$.
The desired result is $\mathcal{H}_\infty = \lim_{n\to\infty} \mathcal{H}_n$.

Theorem 13 (Equivalue for the General Algorithm) *For the general algorithm* $\exists \mathcal{P} \in \mathbb{SOL}^*$ *such that* $(\mathcal{P} - \mathcal{H}) \ominus \mathbb{Z} = \emptyset$ *with* $\mathcal{P} \sim \mathcal{H}_\infty$.

Theorem 14 (Connectivity for the General Algorithm) *For the general algorithm, if* $|models[\mathcal{M}]| \leq 1 \forall \mathcal{M} \in \mathbb{D}_g \cup \mathbb{I}_g$ *then* $|models[\mathcal{H}_{n+1}]| \leq |models[\mathcal{H}_n]| \forall n$ *(i.e. the number of sub-models never increases).*

4.3.2 Minimization Rule

We can clearly see that for the general algorithm for automated code generation it is possible to not be able to obtain suitable model $\mathcal{P} = \mathcal{H}_\infty$ for which $\mathcal{P} \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z} = \emptyset$. One such case is when $\mathcal{H} \ominus (\oplus \mathbb{D}_g) \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z} \neq \emptyset$ and $(\oplus \mathbb{D}_g) \ominus (\oplus \mathbb{I}_g) = (\oplus \mathbb{D}_g)$. Thus, instead of trying to solve a -sometimes- unsolvable problem, we opt for defining a minimization metric $\|\mathcal{H}_\infty \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z}\|$ (i.e. complexity minimization for the non-implementable model segment).

We can see that the above metric fails to take into account the weight for the required solution. To do so, we need to include the quantity $\|\mathcal{P} \ominus (\mathcal{P} \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z})\|$. This quantity needs to be maximized, in order to make selecting a subset of a connected problem sub-optimal to selecting the connected problem itself. Thus, we ensure that there is no missing information. This helps even when the algorithm behaves as a proof-machine, as it produces more relevant axioms to deduct from.

Hence, we introduce a balancing parameter a for which we need to minimize the quantity $\|\mathcal{H}_\infty \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z}\| - a\|\mathcal{P} \ominus (\mathcal{P} \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z})\|$, resulting in the following problem definition;

Definition 19 (Automated Code Generation as a Minimization Problem) Let us assume sets of models $\mathbb{D}_g, \mathbb{I}_g \subseteq \mathbb{SOL}^*$ and function $t_g : \mathbb{SOL}^* \to \mathbb{SOL}^*$ for which;

- $t_g(\mathcal{D}) \sim \mathcal{D} \forall \mathcal{D} \in \mathbb{D}$
- $t_g(\mathbb{D}) = \mathbb{I}$
- $connector\big(\mathcal{D}, t_g(\mathcal{D})\big)$ is known $\forall \mathcal{D} \in \mathbb{D}$

We need to define the function t_g outside $\mathbb{D}$, so that for a given model $\mathcal{H} \in \mathbb{SOL}^*$ exists a model $\mathcal{P} \in \mathbb{SOL}^*$ such that;

$$t_g(\mathcal{H}) \sim \mathcal{P} \tag{24a}$$

$$t_g(\mathcal{H}) \ominus (\oplus \mathbb{I}) \ominus \mathbb{Z} = \emptyset \tag{24b}$$

$$\|\mathcal{P} \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z}\| - a\|\mathcal{P} \ominus (\mathcal{P} \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z})\| = min \tag{24c}$$

Theorem 15 (Minimization Step) *For the sequence* $\{h_n\}_{n \in \mathbb{N}}$ *defined as* $h_n = \|\mathcal{H}_n \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z}\| - a\|\mathcal{H}_n \ominus (\mathcal{H}_n \ominus (\oplus \mathbb{I}_g) \ominus \mathbb{Z})\|$, *applying the general*

algorithm for automated code generation on automated code generation as a minimization problem we obtain;

$$h_n - h_{n+1} = (p_{1,n} - p_{2,n}a)\|\mathcal{D}_n\| + a\|t_g(\mathcal{D}_n)\| \tag{25}$$

where

$$p_{1,n} = \frac{\|\mathcal{H}_n \ominus (\oplus\mathbb{I}_g)\| - \|\mathcal{H}_n \ominus (\oplus\mathbb{I}_g) \ominus \mathcal{D}_n\|}{\|\mathcal{D}_n\|} \tag{26a}$$

$$p_{2,n} = \frac{\|\mathcal{H}_n \ominus (\mathcal{H}_n \ominus (\oplus\mathbb{I}_g))\| - \|\mathcal{H}_n \ominus (\mathcal{H}_n \ominus (\oplus\mathbb{I}_g)) \ominus \mathcal{D}_n\|}{\|\mathcal{D}_n\|} \tag{26b}$$

Proof

We can easily see that $\mathcal{H}_n \ominus \big((\oplus\mathbb{D}_g) \oplus (\oplus\mathbb{I}_g) \oplus \mathbb{Z}\big)$ remains constant for all n. Thus, as it does not contribute to complexity changes, we will conduct this proof without loss of generality for $\mathcal{H}_n \ominus \big((\oplus\mathbb{D}_g) \oplus (\oplus\mathbb{I}_g) \oplus \mathbb{Z}\big) = \emptyset$.

Let us define $\mathbb{I}_e = \mathbb{I}_g \cup \mathbb{Z} = \mathbb{I}_g \cup \{y = x, \forall y\,\emptyset, \exists x\,\emptyset\}$.
Assuming $\mathcal{H}_n \ominus \big((\oplus\mathbb{D}_g) \oplus (\oplus\mathbb{I}_e)\big) = \emptyset$, we obtain $\mathcal{H}_{n+1} \ominus \big((\oplus\mathbb{D}_g) \oplus (\oplus\mathbb{I}_e)\big) = \emptyset$. Therefore, through induction for $\mathcal{H}_0 = \mathcal{H}$, we have proven that $\mathcal{H}_n \ominus \big((\oplus\mathbb{D}_g) \oplus (\oplus\mathbb{I}_e)\big)$ $\forall n$. Now, as $\big(connector(\cdot) \oplus equator(\cdot)\big) \ominus (\oplus\mathbb{I}_e) = \emptyset$ and $\mathcal{H}_{n+1} = \Big(\mathcal{H}_n \ominus \big((\mathcal{H}_n \ominus (\oplus\mathbb{D}_g)\big)\Big) \oplus \Big(\mathcal{H}_n \ominus (\oplus\mathbb{D}_g)\Big)$, we obtain that;

$$\begin{aligned}
\|\mathcal{H}_{n+1} \ominus (\oplus\mathcal{I}_e)\| &= \Big\|\big((\mathcal{H}_n \ominus \mathcal{D}_n) \oplus t_g(\mathcal{D}_n)\big) \ominus (\oplus\mathbb{I}_e)\Big\| \\
&= \Big\|\big((\mathcal{H}_n \ominus \mathcal{D}_n) \ominus (\oplus\mathbb{I}_e)\big) \oplus \big(t_g(\mathcal{D}_n) \ominus (\oplus\mathbb{I}_e)\big)\Big\| \\
&= \Big\|\big(\mathcal{H}_n \ominus \big(\mathcal{H}_n \ominus (\oplus\mathbb{D}_g)\big) \ominus \mathcal{D}_n \ominus (\oplus\mathcal{I}_e)\big) \\
&\quad \oplus \big(\mathcal{H}_n \ominus (\oplus\mathbb{D}_g) \ominus \mathcal{D}_n \ominus (\oplus\mathcal{I}_e)\big) \\
&\quad \oplus \big(t_g(\mathcal{D}_n) \ominus (\oplus\mathcal{I}_e)\big)\Big\| \\
&= \big\|\mathcal{H}_n \ominus \big(\mathcal{H}_n \ominus (\oplus\mathbb{D}_g)\big) \ominus \mathcal{D}_n \ominus (\oplus\mathcal{I}_e)\big\| \\
&\quad + \big\|\mathcal{H}_n \ominus (\oplus\mathbb{D}_g) \ominus \mathcal{D}_n \ominus (\oplus\mathcal{I}_e)\big\| \\
&\quad + \underbrace{\big\|t_g(\mathcal{D}_n) \ominus (\oplus\mathcal{I}_e)\big\|}_{=0} \\
&= \big\|\mathcal{H}_n \ominus \big(\mathcal{H}_n \ominus (\oplus\mathbb{D}_g)\big) \ominus \mathcal{D}_n \ominus (\oplus\mathcal{I}_e)\big\| \\
&\quad + \underbrace{\big\|\mathcal{H}_n \ominus (\oplus\mathbb{D}_g) \ominus (\oplus\mathcal{I}_e)\big\|}_{=0} \\
&= \big\|\mathcal{H}_n \ominus (\oplus\mathcal{I}_e) \ominus \mathcal{D}_n\big\|
\end{aligned}$$

Additionally;

$$\begin{aligned}
\|\mathcal{H}_{n+1} \ominus (\mathcal{H}_{n+1} \ominus (\oplus\mathcal{I}_e))\| &= \Big\| \big((\mathcal{H}_n \ominus \mathcal{D}_n) \oplus t_g(\mathcal{D}_n)\big) \\
&\quad \ominus \big((\mathcal{H}_n \ominus \mathcal{D}_n) \oplus t_g(\mathcal{D}_n) \ominus (\oplus\mathbb{I}_e)\big) \Big\| \\
&= \Big\| \big((\mathcal{H}_n \ominus \mathcal{D}_n) \oplus t_g(\mathcal{D}_n)\big) \\
&\quad \ominus \big((\mathcal{H}_n \ominus \mathcal{D}_n) \ominus (\oplus\mathbb{I}_e)\big) \Big\| \\
&= \Big\| (\mathcal{H}_n \ominus \mathcal{D}_n) \ominus \big((\mathcal{H}_n \ominus \mathcal{D}_n) \ominus (\oplus\mathbb{I}_e)\big) \Big\| + \big\| t_g(\mathcal{D}_n) \big\| \\
&= \Big\| (\mathcal{H}_n \ominus \mathcal{D}_n) \ominus \big(\mathcal{H}_n \ominus (\oplus\mathbb{I}_e)\big) \Big\| + \big\| t_g(\mathcal{D}_n) \big\| \\
&= \Big\| \mathcal{H}_n \ominus \big(\mathcal{H}_n \ominus (\oplus\mathbb{I}_e)\big) \ominus \mathcal{D}_n \Big\| + \big\| t_g(\mathcal{D}_n) \big\|
\end{aligned}$$

Also, $\|\mathcal{H}_{n+1} \ominus (\oplus\mathcal{I}_e)\| = \|\mathcal{H}_n \ominus (\oplus\mathcal{I}_e)\| + \|t_g(\mathcal{D}_n)\| - p_{1,n}\|\mathcal{D}_n\|$
and $\|\mathcal{H}_{n+1} \ominus (\mathcal{H}_{n+1} \ominus (\oplus\mathcal{I}_e))\| = \Big\|\mathcal{H}_n \ominus \big(\mathcal{H}_n \ominus (\oplus\mathbb{I}_e)\big)\Big\| - p_{2,n}\|\mathcal{D}_n\| + \|t_g(\mathcal{D}_n)\|$.
Thus, $h_n = \|\mathcal{H}_n \ominus (\oplus\mathcal{I}_e)\| - a\|\mathcal{H}_n \ominus (\mathcal{H}_n \ominus (\oplus\mathcal{I}_e))\|$, from which we obtain;
$\boxed{h_n - h_{n+1} = (p_{1,n} - p_{2,n}a)\|\mathcal{D}_n\| + a\|t_g(\mathcal{D}_n)\|}$.

4.4 Equivalue Implementation Choices

Proposition 1 (Quantifier Implementation) *For variables x, y we assume that $\forall x\, \emptyset$ is equivalue to declaring a function input and $\exists y\, \emptyset$ is equivalue to returning the value of y within a function.*

Proposition 2 (Equivalue Implementation) *Two variable sets $\mathcal{V}_1, \mathcal{V}_2$ of formulas e_1, e_2 respectively will be considered equivalue if $|\mathcal{V}_1| = |\mathcal{V}_2|$ (i.e. they contain the same number of variables) and, denoting the $1-1$ variable replacement as @, it also holds that $e_1@\mathcal{V}_2 = e_2$ and $e_2@\mathcal{V}_1 = e_1$.*

Proposition 3 (Equivalue from Lexicographic Similarity) *For any set of formulas $\mathbb{D}$ which does not contain blocks of code, exists a constant d for which $\forall m, n \in \mathbb{D}$ it holds that;*

$$\Big[\big|\mathcal{V}[m]\big| = \big|\mathcal{V}[n]\big|, lex(m,n) \geq d\Big] \Rightarrow m \sim n \tag{27a}$$

$$connector(m,n) = \{\mathcal{V}[m] = \mathcal{V}[n]\} \tag{27b}$$

This assumption is quite loose for predicates with a lot of arguments, although for correct flattening it is reduced to single predicate matching. It is important that we use lexicographic similarity and not some other type of bag-of-words similarity (such as cosine similarity), as it actually allows the above assumption to satisfy all conditions for equivalue.

Proposition 4 (Blocks of Code and Comments) *Each block of code can be considered a multi-variable multi-output function. Comment predicates with unknown structure can be considered a single predicate (i.e. using variables alongside those comments results in depth of 2).*

4.5 Greedy Choice and Optimality

Theorem 16 (Average Minimization Step under Implementation Choices) *For equivalue from lexicographic similarity and equivalue implementation, for the minimization step, it holds that;*

$$mean(p_{1,n}) = 1 \tag{28}$$

If, additionally, all formulas in $\big((\oplus\mathbb{D}_g) \oplus (\oplus\mathbb{I}_g) \oplus (\oplus\{\mathcal{H}_n\}_{n\in\mathbb{N}})\big) \ominus \mathbb{Z}$ *are flattened, then*

$$mean(p_{2,n}) \leq 1 \tag{29}$$

Proof
Similarly to minimization step, it suffices to show the desired result for $\mathcal{H}_n \ominus \big((\oplus\mathbb{D}_g) \oplus (\oplus\mathbb{I}_g) \oplus \mathbb{Z}\big) = \emptyset$.

It also suffices to show that for each random pair of models $\mathcal{H}_1, \mathcal{D}_1$ with $p_{1,n} \neq 0$ exists a single pair $\mathcal{H}_2, \mathcal{D}_2$ such that $\sum_{i=1,2} \frac{\|\mathcal{H}_i\| - \|\mathcal{H}_i \ominus \mathcal{D}_i\|}{\|\mathcal{D}_i\|} = 2$.

Let us now define the maximum model $\mathcal{S} \subseteq \mathcal{H}_1$ (it may *not* be a sub-mode) such that $(\mathcal{H}_1 \ominus \mathcal{S}) \ominus (\mathcal{D}_1) = \mathcal{H}_1 \ominus \mathcal{S}$. As $p_{1,n} \neq 0$, $\mathcal{S} \neq \emptyset \Leftrightarrow \|\mathcal{S}\| \neq 0$.

We proceed to generate the desired models as $\mathcal{H}_2 = (\mathcal{H}_1 \ominus \mathcal{S}) \oplus \mathcal{D}_1$ and $\mathcal{D}_2 = \mathcal{S}$ (i.e. we exchange formulas that cause the fraction to diverge from 1).

Thus, we obtain $\mathcal{S} \sim \mathcal{D}_1$, $\mathcal{H}_1 \ominus \mathcal{D}_1 = \mathcal{H}_1 \ominus \mathcal{S} = \mathcal{H}_2 \ominus \mathcal{S}$ and, as $\mathcal{S}$ was selected to be a maximum model,
$\|\mathcal{H}_1 \ominus \mathcal{S}\| + \|\mathcal{S}\| = \|\mathcal{H}_1\| - \|\mathcal{S}\| + \|\mathcal{S}\| = \|\mathcal{H}_1\|$.
Also, $\|\mathcal{H}_2 \ominus \mathcal{D}_1\| + \|\mathcal{D}_1\| = \|\big((\mathcal{H}_1 \ominus \mathcal{S}) \oplus \mathcal{D}_1\big) \ominus \mathcal{D}_1\| + \|\mathcal{D}_1\| = \|(\mathcal{H}_1 \ominus \mathcal{S}) \ominus$

$\mathcal{D}_1\| + \|\mathcal{D}_1\| = \|\mathcal{H}_1 \ominus \mathcal{S}\| + \|\mathcal{D}_1\| = \|(\mathcal{H}_1 \ominus \mathcal{S}) \oplus \mathcal{D}_1\| = \|\mathcal{H}_2\|$. Therefore;

$$
\begin{aligned}
2 = \frac{\|\mathcal{H}_1\| - \|\mathcal{H}_1 \ominus \mathcal{D}_1\|}{\|\mathcal{D}_1\|} + \frac{\|\mathcal{H}_2\| - \|\mathcal{H}_2 \ominus \mathcal{D}_2\|}{\|\mathcal{D}_2\|} \\
\Leftrightarrow \|\mathcal{H}_1\|\|\mathcal{S}\| + \|\mathcal{H}_2\|\|\mathcal{D}_1\| \\
- \|\mathcal{H}_1 \ominus \mathcal{D}_1\|\|\mathcal{S}\| - \|\mathcal{H}_2 \ominus \mathcal{S}\|\|\mathcal{D}_1\| = 2\|\mathcal{D}_1\|\|\mathcal{S}\| \\
\Leftrightarrow \|\mathcal{H}_1 \ominus \mathcal{S}\|\|\mathcal{S}\| + \|\mathcal{S}\|^2 + \|\mathcal{H}_2 \ominus \mathcal{D}_1\|\|\mathcal{D}_1\| + \|\mathcal{D}_1\|^2 \\
- \|\mathcal{H}_1 \ominus \mathcal{D}_1\|\|\mathcal{S}\| - \|\mathcal{H}_2 \ominus \mathcal{S}\|\|\mathcal{D}_1\| = 2\|\mathcal{D}_1\|\|\mathcal{S}\| \\
\Leftrightarrow \|\mathcal{H}_1 \ominus \mathcal{D}_1\|\|\mathcal{S}\| + \|\mathcal{S}\|^2 + \|\mathcal{H}_2 \ominus \mathcal{D}_1\|\|\mathcal{D}_1\| + \|\mathcal{D}_1\|^2 \\
- \|\mathcal{H}_1 \ominus \mathcal{D}_1\|\|\mathcal{S}\| - \|\mathcal{H}_2 \ominus \mathcal{D}_1\|\|\mathcal{D}_1\| = 2\|\mathcal{D}_1\|\|\mathcal{S}\| \\
\Leftrightarrow \|\mathcal{S}\|^2 + \|\mathcal{D}_1\|^2 = 2|\mathcal{D}_1\|\|\mathcal{S}\| \\
\Leftrightarrow (\|\mathcal{S}\| - \|\mathcal{D}_1\|)^2 = 0 \\
\Leftrightarrow \|\mathcal{S}\| = \|\mathcal{D}_1\|
\end{aligned}
$$

However, $\|\mathcal{S}\| = \|\mathcal{D}_1\|$ actually holds true, as $\mathcal{S} \sim \mathcal{D}_1$ and $\mathcal{S} \ominus \mathbb{Z}, \mathcal{D}_1 \ominus \mathbb{Z} \in \mathbb{D}_g$. Also, thanks to the construction process, this mapping from $\mathcal{H}_1, \mathcal{D}_1$ to $\mathcal{H}_2, \mathcal{D}_2$ is invertible. Thus, $mean_{p_{1,n} \neq 0}(p_{1,n}) = 1$. However, as $\mathcal{H}_n \ominus \mathcal{D}_n \neq \mathcal{H}_n$ in the general algorithm for automated code generation, we obtain that $p_{1,n} \neq 0$ always. Thus, $mean(p_{1,n}) = 1$.

Similarly, by substituting $\mathcal{H}_n \leftarrow \mathcal{H}_n \ominus (\mathcal{H}_n \ominus (\oplus \mathbb{I}_g))$, Equivalue Implementation Choices theorems yield, $mean_{p_{2,n} \neq \emptyset}(p_{2,n}) = 1 \Rightarrow mean(p_{2,n}) \leq 1$.

Theorem 17 (Average Case Greedy Method) *In the average case, the greedy method for selecting the appropriate $\mathcal{D}_n$ is;*

$$(1 - a')\|\mathcal{D}_n\| + a'\|t_g(\mathcal{D}_n)\| = max \tag{30}$$

for each step in the general algorithm for automated code generation, where;

$$a' = \frac{a}{1 + a - a\, mean(p_{2,n})} \tag{31}$$

(thus $a' \leq a$ for flattened formulas).

Proof
Detrimental, by applying q-complexity for equivalue implementation on minimization step.

This optimizes our algorithm's execution speed, as we can pre-order the set $\mathbb{D}_g$ to get binary search times. We can even use mixed strategies, such as using binary search and then perform the numerically exact minimization step in the local area of the binary result.

Theorem 18 (General Algorithm Optimality) *By arbitrarily selecting the model $\mathcal{D}_n \in \mathbb{D}_g$ such that $\mathcal{H}_n \ominus \mathcal{D}_n \neq \emptyset$ in each step of general algorithm for automated code generation, the probability of reaching a global minimum for automated code generation as a minimization problem monotonically approaches* 1 *as $p_{\mathbb{D}_g \ominus \mathbb{Z}}\big((\mathbb{D}_g \cup \mathbb{I}_g) \ominus \mathbb{Z}\big)$ approaches* 1*.*

Despite the optimality under the favorable condition $p_{\mathbb{D}_g \ominus \mathbb{Z}}\big((\mathbb{D}_g \cup \mathbb{I}_g) \ominus \mathbb{Z}\big) = 1$, the condition itself is extremely strict, as it essentially requires no overlapping between problem descriptions or between any description and any implementation. This condition is even more difficult to adhere to in case of flattening.

Theorem 19 (Global Minimum Existence) *For the general algorithm for automated code generation exists a global minimum with probability that approaches* 1 *as $p_{\mathbb{D}_g}(\mathbb{I}_g)$ approaches* 1*.*

The condition $p_{\mathbb{D}_g}(\mathbb{I}_g) = 1$ is an easier one to uphold and fails only when trying to use our algorithm as a proof machine (with code formulas present in the comments).

Theorem 20 (Average Case Greedy Method Finiteness for Flattened Data) *As long as a global minimum exists, in the average case, using either greedy method on minimization step or average case greedy method, the general algorithm for automated code generation converges to a value for automated code generation as a minimization problem $\forall a' \in [0, 1)$ within a* finite number of steps*, as long as $\mathcal{H}, \mathbb{D}_g, \mathbb{I}_g$ are flattened.*

Proof
If $\mathcal{H}, \mathbb{D}_g, \mathbb{I}_g$ are flattened, it follows that $\mathcal{H}_n$ is also flattened $\forall n$. Thus, according to average minimization step under implementation choices, we obtain $mean(p_{2,n}) = 1$ and $mean(p_{2,n}) \leq 1$.
In the average case, selecting the maximization of h_{n+1} in either minimization step or average case greedy method makes the sequence $\{h_n\}$ monotonically decreasing in the average case, with $mean(h_n - h_{n+1}) \geq mean(p_{1,n} - p_{2,n}a) = mean(p_{1,n}) - mean(p_{2,n})a \geq 1 - a$. If $a' < 1 \Rightarrow a < 1$, the decrease step is a positive constant. Therefore, h_n must necessarily converge to value less than or equal to the minimum within a finite number of steps.

Theorem 21 (Greedy Method on Minimization Step Optimality) *Let $\mathcal{H} \in \mathbb{SOL}^*$ be a model on which the greedy method for minimization step on general algorithm for automated code generation has converged to $\mathcal{H}_\infty$. The probability of the converged value being a global minimum approaches* 1 *as $\|\mathbb{D}_g\|_{\mathbb{D}_g}$ approaches infinity.*

Theorem 22 (Average Case Optimality) *Let $\mathcal{H} \in \mathbb{SOL}^*$ be a model on which average case greedy method on general algorithm for automated code generation has been applied and converged to $\mathcal{H}_\infty$. The probability of the converged value being a global minimum approaches* 1 *as $\|\mathbb{D}_g\|_{\mathbb{D}_g}$ approaches infinity.*

These two conditions are extremely strong, as they basically state that, for a large enough non-repeatable database ($\|\mathbb{D}_g\|_{\mathbb{D}_g} \to \infty$), the converged value is a global minimum with enough certainty. Note that converging to infinity does not necessarily imply the inclusion of helpful models; according to the way the database grows, a problem may be kept unsolvable. However, we become increasingly certain for the optimality of our solution for larger databases.

4.6 Final Algorithm

According to the previously stated theorems, we obtain the following greedy algorithm for automated code generation:

$\big[\mathbb{D}_g \sim t_g(\mathbb{D})\big] \leftarrow independent\Big(flattened\Big(\cup_{\mathcal{D}\in\mathbb{D}}\big[problems[\mathcal{D}] \sim t(problems[\mathcal{D}])\big]\Big)\Big)$
$\mathcal{H}_0 \leftarrow \mathcal{H}$
$n \leftarrow 0$
while ($\mathcal{H}_n$ not implementable) and ($\mathcal{D}_n \neq \emptyset$) **do**
  $n \leftarrow n+1$
  find $\mathcal{D}_n \in \mathbb{D}_g$ such that $\mathcal{H}'_n \ominus \mathcal{D}_n \neq \emptyset$ and $(1-a')\|\mathcal{D}_n\| + a'\|t_g(\mathcal{D}_n)\|$ (or the minimization step) is minimized
  select an arbitrary $\mathcal{H}'_n \in models[\mathcal{H}_n]$ such that $\mathcal{H}'_n \ominus (\mathcal{H}'_n \ominus \mathcal{D}_n) \neq \emptyset$
  $\mathcal{C} \leftarrow connector\big(\mathcal{H}'_n, \mathcal{D}_n\big) \oplus connector\big(\mathcal{D}_n, t_g(\mathcal{D}_n)\big) \oplus equator(\mathcal{D}_n, \mathcal{H}'_n)$
  $\mathcal{H}'_{n+1} \leftarrow independent_{\mathbb{D}_g}\Big((\mathcal{H}'_n \ominus \mathcal{D}_n) \oplus t_g(\mathcal{D}_n) \oplus \mathcal{C}\Big)$
  $\mathcal{H}_{n+1} \leftarrow \Big(\oplus\big(models[\mathcal{H}_n] - \mathcal{H}'_n\big)\Big) \oplus \mathcal{H}'_{n+1}$
end while
$\mathcal{H}_\infty = \mathcal{H}_n$

The first step generates sub-models, in order to ensure connectivity of models $\mathcal{D}_n$ and thus retain connectivity, resulting in $|models[\mathcal{H}_\infty]| \leq |models[\mathcal{H}]|$. Also, the algorithm is essentially performed discretely to each member of $models[\mathcal{H}_n]$, so as to finally ensure that $|models[\mathcal{H}]| = |models[\mathcal{H}_\infty]|$. To check for non-implementability, we use the condition $\mathcal{H}_n \ominus (\oplus\mathbb{I}_g) \ominus \mathbb{Z} \neq \emptyset$ and will later restrict that models in $\mathbb{I}$ (and thus in $\mathbb{I}_g$) can only have formulas that are;

a Equations between variables and formulas.

b Blocks of code (containing any possible flow control checks).

c Declarations of inputs or return values (according to quantifier implementation, these are quantifiers on the empty formula; $\forall x\, \emptyset$ and $\exists y\, \emptyset$).

Flattening also takes place, in order to ensure finiteness when a global minimum exists, thanks to average case greedy method finiteness for flattened data.

4.7 Implementing $\mathcal{H}_\infty$

Our approach on *Automated Code Generation* using Second-Order Logic produces a model $\mathcal{H}_\infty$, which is essentially an unordered set of formulas. Those formulas simultaneously adhere to a set of constraints, specified by applying the developed theory on the original model $\mathcal{H}$. $\mathcal{H}_\infty$ is known as a Logic Program.[59] Unfortunately, this method fails to actually implement the model it in a programming language; due to not being ordered. Even if individual formulas are implementable, there exists a certain uncertainty on the *order* they should be placed within a sequential program. Of course, for independent formulas (i.e. with no common variable) their ordering may be inconsequential, but for purely sequential logic on solving a problem, this may prove detrimental.

To resolve this situation, we order the formulas of $\mathcal{H}_\infty$ according to their assignment dependency; that is, we make formulas which contain assignment to variables always precedent any further occurrences of those variables. This method only falls short of handling sequential code that re-assigns different values to the same variable later on, so we also retain the same ordering as within the model $t_g(\mathcal{D}_n)$. This covers most code generation needs, but further study is needed to cover cases of circular logic. For a more formalized approach on this, we could use traversal variables.[48] Alternatively, we could directly use existing frameworks[60] to generate solution sets for the generated Logic Program.

Before $\mathcal{H}_\infty$ implementation, one could also perform partial un-flattening[53] by using assignments between variables and formulas to directly replace variable occurrences in greater (as specified by ordering) blocks of code. This way, naturalness (i.e. comprehension by humans) can be preserved in produced code, as long as implementations in $\mathbb{I}_g$ are also natural. To retain equivalue, empty quantifiers should be kept throughout this procedure. This is hypothesised to vastly improve intelligibility of produced code, although a systematic approach could use more sophisticated criteria for replacing variable occurrences in a way that ensures naturalness.

5 Database Import

5.1 Code and Comments

Proposition 5 (Code Comments) *For a block of code, its comments are those that are placed either inside or before it but not inside another same-level block.*

Proposition 6 (Flow Control) *Flow-control structures are contained in the block of code they control.*

Proposition 7 (Comment Splitting) *It is possible to split comments into smaller, independent models.*

In the simplest case, comment splitting can be performed on certain predicates (such as *and*). In more advanced forms, we could also take syntax into account.

Proposition 8 (Equivalence between Code and Comments) *For a complete model $\mathcal{C} \in \mathbb{SOL}^*$ which contains a function comments and $t(\mathcal{C}) \in \mathbb{SOL}^*$ the code of that function (with inputs and possible outputs quantified according to quantifier implementation, but return statements not actually replaced), we assume that;*

$$\Big[\forall x \in \big(\mathcal{V} - \mathcal{X}\big)\big[t(\mathcal{C})\big]\, x \in \mathcal{V}[\mathcal{C}]\Big] \Leftrightarrow [t(\mathcal{C}) \leftrightarrow \mathcal{C}] \tag{32}$$

The above proposition states that a function's comments are equivalent to code as long as all input and output variables are contained in the comments. However, there are cases where this requirement is not met. These are;

a If the model $\mathcal{C}$ is not complete.

b If return statements return formulas instead of single variables.

c If comments donnot contain all variables linked to quantifiers.

We devise some naive solutions for these problems in Comment Transformations for Variables.

5.2 Variable Identification

In equivalence between code and comments we arbitrarily assumed that we have a way for identifying which predicates are actually variable names. However, function code formulas are purely alpharithmetic strings that oftenly call functions residing in external libraries. In typed languages, variable identification is quite easy, as variables must be declared. For untyped languages though, we need a more sophisticated method, depending on language structure. One such method for languages that use the equality operator is the following;

Theorem 23 (Variable Recognition in Second-Order Logic with Equality Operator) *Let us assume a language with equality operator and a set of variable types $\mathcal{T}$. In that language, a model that does not use global variables contains only the following variables;*

- *all variables of its parent block*
- *all variables of blocks within the same parent block*

- *all words that directly follow a quantifier but donnot include operators, parenthesis or brackets (inputs and return statements are considered quantifiers, according to quantifier implementation)*
- *all words k whose block conforms according to Regular Expressions to;*

$$(N|;|\square)^+ (e|TYPE|THIS)\, k\, \square^* =$$

where $\square$ indicates space character, N the new line character, ; the command end character, $TYPE$ is any data type and $THIS$ an object reference predicate - the object reference predicate is always e for procedural programming.

Function implementation blocks, along with function arguments, are considered to be one block lower than function declarations.

For untyped languages, the above theorem can be applied easily, as the set of type predicates is empty. Also, this theorem can be extended to include variables in superior blocks of code, depending on language structure. For higher-level object-oriented languages (such as Java), applying the theorem is a more complex procedure, as the object reference predicate $THIS$ (i.e. "$this$.") may be implied due to variable range. Python is more strict in that aspect, making all uses of the "$self$." predicate mandatory.

Theorem 24 (Object Reference Variable) *To interpret code logic in all possible ways, we must generate a different model for each combination of the binary choices of considering the object-reference predicate as part of a variable name or an independent variable.*

However, in most cases it suffices to generate two models; one for using the object-referencing predicate as part of the variable name and one for considering it a separate variable. Note that, although flattening actually allows for all such interpretations, our inflexible equivalue from lexicographic similarity and equivalue implementation (which require the same number of variables) make the use of separate models mandatory, as we want equivalence between code and comments to correspond to logical integrity.

5.3 Comment Transformations for Variables

To circumvent the problems that arise from our accepted equivalence between code and comments, we use a number of simple strategies which, although not necessarily universally optimal, still achieve sufficient results;

a If the model $\mathcal{C}$ is not complete (i.e. if there are formulas which donnot contain variables), we can assume that all comment formulas without variables refer to the corresponding code as the whole. Therefore, we can append names of variables in corresponding code at the end of those comments.

b For returned formulas, we just need to create additional variables to assign the formula to and then return them.

c In cases where comments donnot contain variables linked to quantifiers (the previous case always results to this scenario), we can generate equalities between input and output variables. This work-arround only solves the problems for functions that perform simple transformations. In more complex cases, the problem of matching the correct input with the correct output is prevalent and needs additional analysis. However, we can reasonably assume that the first inputs is the one transformed, as programming languages encourage placing default (and thus non-vital) arguments last in the argument list.

5.4 Equivalue Extension

In this paragraph, we extend equivalue even further. The following definition still retains all equivalue properties, while introducing concepts prevalent in human speech. This way, we are able to identify a broader and more abstract context range;

Definition 20 (Equivalue Extension) We define the following properties to extend equivalue, using Regular Expressions and denoting the space character as $\square$;

- We define sets of strings $\mathcal{A}, \mathcal{B}$ and a $1-1$ mapping between them inv : $\mathcal{B} \to \mathcal{A}$ for which $[b = inv(a), x \sim m, y \sim n] \Rightarrow x\square a\square y \sim n\square b\square m$.
- We define sets of strings $\mathcal{C}, \mathcal{D}$ with $\mathcal{C} \cap \mathcal{D} = \emptyset$ and function $same : \mathcal{C} \to \mathcal{D}$ for which $[c \sim same(d), x \sim m, y \sim n] \Rightarrow x\square c\square y \sim m\square d\square n$ if neither expression is a block of code.
- We define a set of endings $\mathcal{E}$ for which $[x \sim m, y \sim n] \Rightarrow x\square pe_1\square y \sim m\square pe_2\square n \, \forall e_1, e_2 \in \mathcal{E}, p \notin \mathcal{A} \cap \mathcal{B}$ if neither expression is a block of code.
- We define a set of special characters $\mathcal{P}$ for which $[x \sim m, y \sim n] \Rightarrow x\square p\square y \sim mpn \, \forall p \in \mathcal{P}$.
- We define that $[x \sim m, y \sim n] \Rightarrow x\square^{+} y \sim x\square y$ if neither expression is a block of code.
- We define that $x \sim m \Rightarrow \square x\square \sim m$.
- We define a set of strings to ignore $\mathcal{G}$ for which $[x \sim m, y \sim n] \Rightarrow x\square g\square y \sim x\square y \, \forall g \in \mathcal{G}$ if neither expression is a block of code.

To implement the above extension in the Python language, we can transform each imported formula t in the following way;

$t \leftarrow \square t\square$

if $:\in t$ **then**
$t \leftarrow t.replace(\square ke\square, \square k\square)\, \forall e \in \mathcal{E}, k \in \mathbb{SOL}$
$t \leftarrow t.replace(\square d\square, \square same(d)\square)\, \forall d \in \mathcal{D}$
$t \leftarrow t.replace(p_1\square b\square p_2, p_2\square inv(b)\square p_1)\, \forall b \in \mathcal{B}$
end if
$t \leftarrow t.replace(\square^+ g\square^+, \square)\, \forall g \in \mathcal{G}$
$t \leftarrow t.replace(\square^+, \square)$
$t \leftarrow t.trim()$

where $\square$ is the space character. Note that the condition $:\in t$ implements the detection of a block of code in Python.

6 Concept Implementation

We have developed the *Code Analyzer* application as a real-world application of discussed concepts. It implements all outlined usability, such as importing from libraries and performing the Final Algorithm using greedy method on minimization step. Implementation is multi-threaded, allowing for really fast code generation.

Code Analyzer was developed in Java. However, automatically generated code is produced in Python, as it presents a wide range of favorable characteristics discussed throughout this paper. For coherent and intuitive user interaction, models are called problems.

6.1 Automated Database Import

To import data from software libraries, *Code Analyzer* uses all assumptions and theorems outlined in section Database Import. To be more precise, it detects all functions within the library, alongside with their comments. There is a user interaction to approve the import of detected problems (i.e. models and their implementations), as shown in figure Figure 1. Already existing problems are pre-selected to not be imported, so as to not accidentally overwrite altered ones. For management purposes, each problem is assigned to an in-program library. This library is by default the one being imported, but this can later be altered.

To assist the importing process, comment Notation is also supported. This is imperative for detecting variables without assignment that are neither inputs nor outputs.

Imported problems and their implementations are opened in separate tabs, as shown in figure Figure 2. We can observe the application of equivalue extension, where smaller font indicates ignored parts. Rules for equivalue extension

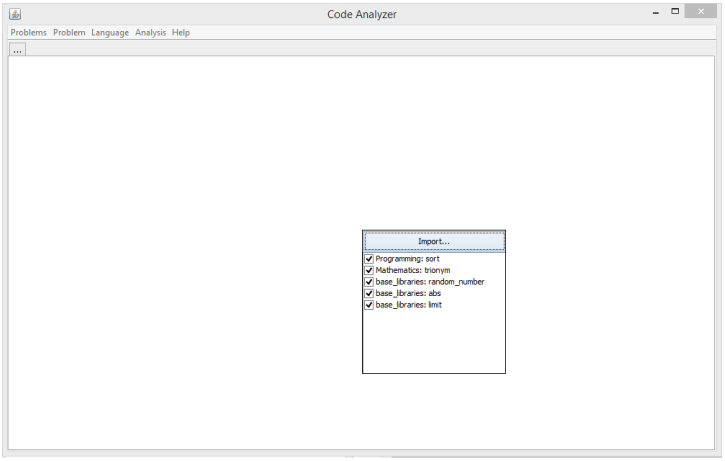

Fig. 1 Database Import

are applied on runtime rather than import time, allowing for flexibility and experimentation on altering relevant sets through the *Language* menu.

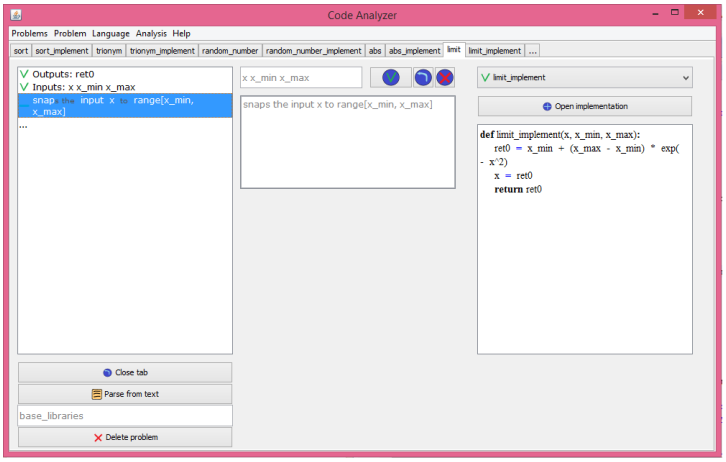

Fig. 2 Database Import Result

6.2 Manual Problem Generation

Besides automated database import, it is also possible to manually generate problems within the *Code Analyzer* environment. It is also possible to alter already existing problems.

Problems→Create and Open generates and opens a new empty problem. Inserting formulas can be done either by writing free text or by generating each individual formula. For text import, there exists appropriate Notation for declaring inputs, outputs, other variables or changing a particular aspect of the problem.

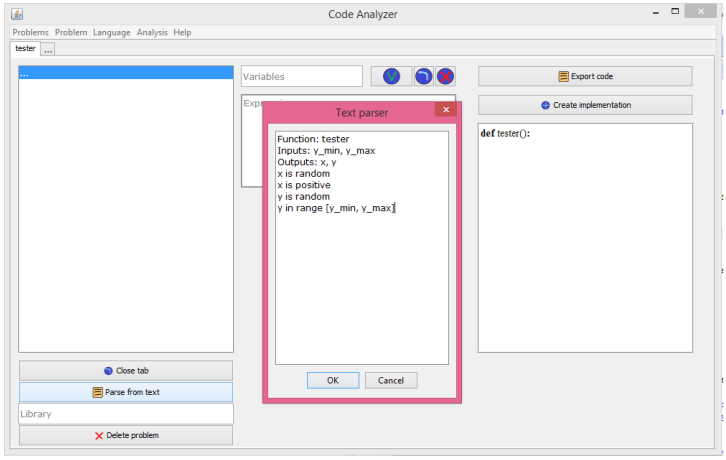

Fig. 3 Manual Problem Generation

Code Analyzer allows manual formula editing and insertion. After editing a formula, changes should be committed (by clicking on the appropriate button). To insert a new empty formula, select .... Manually inserting each formula is more reliable, but it requires the specific declaration of variables for each statement. It is possible to perform automatic variable detection by selecting *Problem→Auto Correct*. It is also possible to observe other discussed transformations, such as flattening (by selecting *Problem→Expand*) and sub-model construction (by selecting *Problem→Subproblem Analysis*).

Problem library can also be manually changed and the problem's implementation can be set amongst problems within the same library.

6.3 Notation

To specifically declare function name, library name or variables, the following notation can be used;

- *Function* : *name* sets the problem name as *name* (it impacts automated database import *only*)
- *Library* : *lib* sets the problem library as *lib*

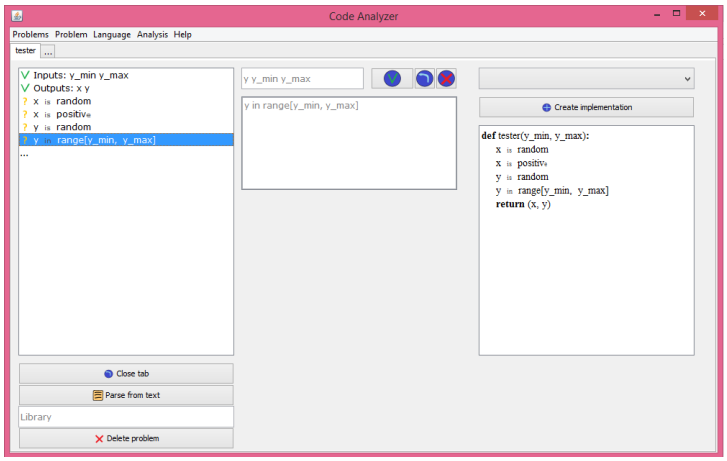

Fig. 4 Manual Problem Generation- Individual Formula Editing

- *Inputs* : $in_1, in_2, ...$ declares $in_1, in_2, ...$ as input variables
- *Outputs* : $out_1, out_2, ...$ declares $out_1, out_2, ...$ as output variables
- $Variables$: $var_1, var_2, ...$ declares $var_1, var_2, ...$ as variable names (input and output variables donnot need to be declared this way)
- By using { and }, separate formulas can be grouped into single blocks of code.

Notation is *not* case sensitive and commas can be freely replaced by spaces.

6.4 Automated Code Generation

Code Analyzer also implements our Final Algorithm. There are several parameters that impact execution;

- **Max iterations**
  The maximum number of iterations of minimization step (so as to reach in exit points, in case a global minimum does not exist).
- **Conserve**
  Corresponds to the parameter a.
- **Similarity**
  Corresponds to constant d for equivalue from lexicographic similarity.

Algorithm implementation is Multi-Threaded, selecting the maximum available number of threads for JVM. It splits the set $\mathbb{D}_g$ amongst threads and compares the best results amongst threads to greedily select $\mathcal{D}_n$. In case of collisions, user decisions are required. Generation of $\mathbb{D}_g, \mathbb{I}_g = t_g(\mathbb{D}_g)$ from

$\mathbb{D}, \mathbb{I} = t(\mathbb{D})$ is also Multi-Threaded. Also, certain libraries can be omitted to save execution time.

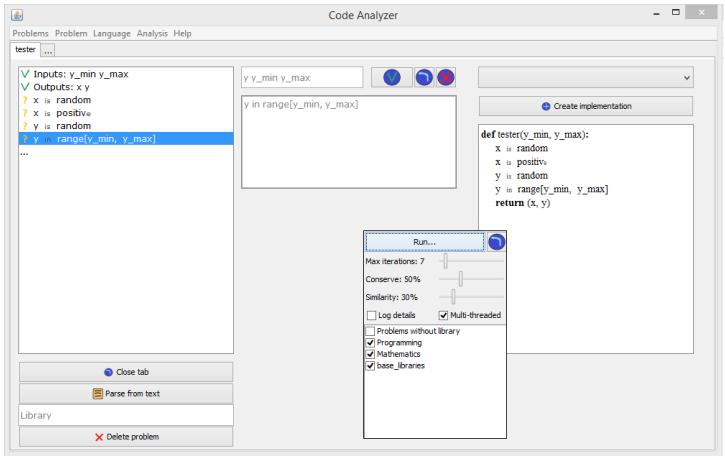

Fig. 5 Options

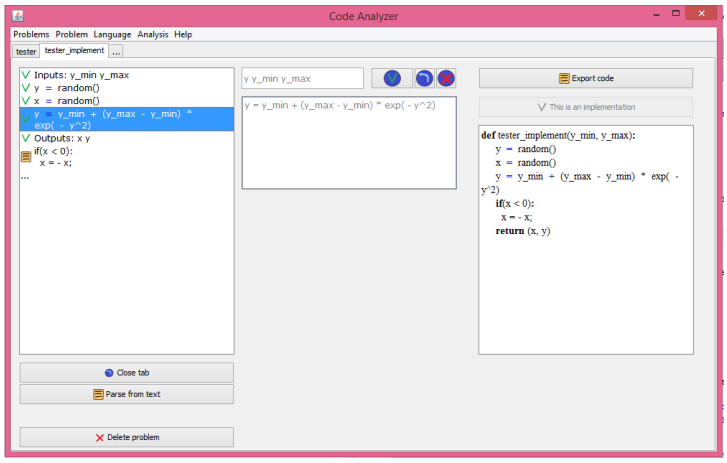

Fig. 6 Results

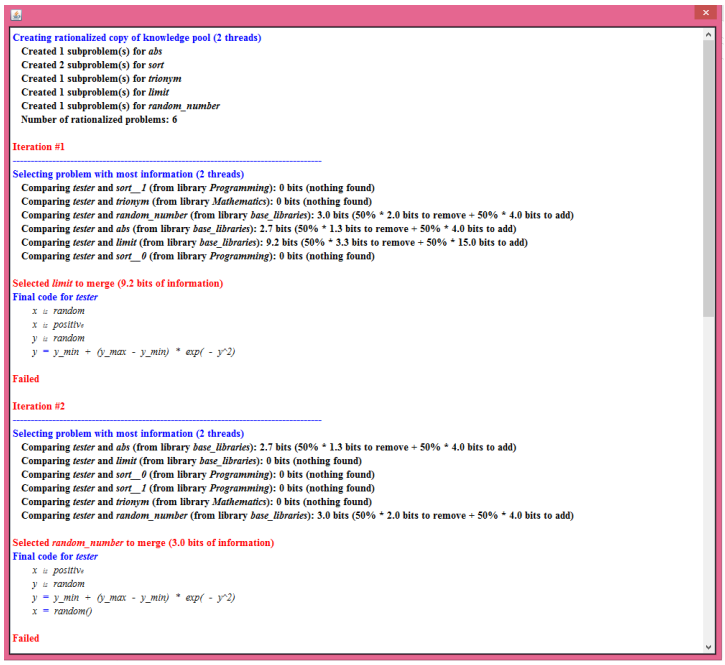

Fig. 7 Details

6.5 Experimental Results

Code Analyzer successfully solves simple well-stated problems. Take, for example, the following problem;

```
Function: test1
Inputs: m, n, k
Outputs: h, b
Variables: a, c
trionym with coefficients m n k
trionym roots h b
((a + b + c) + d) = 1
```

Its implementation is automatically generated as;

```
def test1(m, n, k):
    b = None;
    h = None;
    ((a + b + c) + d) = 1
    D = n^2 - 4 * m * k;
    if(D >= 0):
        h = ( - n + D  **  0.5) / (2 * m);
        b = ( - n - D  **  0.5) / (2 * m);
```

```
    return (h, b)
```

No relevant implementation as found for the formula $((a+b+c)+d)=1$ and thus it remains unsolved.

Let us now observe an example of combining formulas across different problems;

```
Function: test2
Inputs: x_min, x_max
Outputs: y, x
y is random
y is positive
x is random
x in range[x_min, x_max]
```

There exist two sub-models in the above problem and we can see that they truly are solved independently;

```
def test2(x_min, x_max):
    x = random()
    y = random()
    x = x_min + (x_max - x_min) * exp( - x ^ 2)
    if(y < 0):
       y = - y;
    return (y, x)
```

For the above examples, the following library had been imported;

```
"""
Library: Programming
For any: k
b_desc[k]>=b_desc[k+1]
b_asc[k]<=b_asc[k+1]
b_desc rearranges a
b_asc rearranges a
"""
def sort(a):
   # sort a into b_desc in descending order
   b_desc = a.copy();
   for i in range(0, len(a)):
      for j in range(i + 1, len(a)):
         if(b_desc[i] < b_desc[j]):
            b_desc[i], b_desc[j] = b_desc[j], b_desc[i];
   # sort a into b_asc in ascending order
   b_asc = a.copy();
   for i in range(0, len(a)):
```

```
      for j in range(i + 1, len(a)):
         if(b_asc[i] > b_asc[j]):
            b_asc[i], b_asc[j] = b_asc[j], b_asc[i];
   return b_desc, b_asc

"""
Library: Mathematics
trionym coefficients a, b, c
trionym roots x1, x2
"""
def trionym(a, b, c):
   # calculate trionym determinant D
   x1 = None;
   x2 = None;
   # roots exist only if D>=0
   if((D=b^2-4*a*c)>=0):
      x1 = (-b+D^0.5)/(2*a);
      x2 = (-b-D^0.5)/(2*a);
   return x1, x2

def random_number():
   # this function returns a random number
   return random()

def abs(x):
   # this function calculates the absolute value of a given number
   # the absolute value of a number is always positive
   if(x<0):
      x = -x;
   # x>=0
   return x;

def limit(x, x_min, x_max):
   # snaps the input x to range [x_min, x_max]
   return x_min+(x_max-x_min)*exp(-x^2);
```

7 Conclusion

This work presents a breadth of tools that can be used on any automated reasoning scope based on Second Order Logic. To start with, q-complexity is a metric for measuring how *complex* a given Second-Order Logic formula or model is. It can thus be used as a base to define new metrics in varying situations, like we do for equivalueless q-complexity between sets of models.

Furthermore, given any suitable equivalue definition, we are able to use condition application to either perform generalizations or merge logical entities while retaining equivalue. Thankfully, inverse algebrization ensures that such a process can also produce two-way implications. Also, note that other definitions of an equivalue relation may correspond meanings other than programmatic equivalence. But even then, we can still use condition application to infer results with processes similar to general algorithm for automated code generation.

Having established those tools, we define automated code generation and expand this definition in a way friendlier to those tools in loose automated code generation. We also present a general algorithm for automated code generation, which establishes a framework for solving the latter while retaining properties, such as equivalue and connectivity. Afterwards, we further enhance our definition, turning it into a minimization problem and even calculating the exact minimization step. We then present some reasonable assumptions on how to proceed for developing a suitable algorithm and show that they lead to average minimization step under implementation choices presenting favorable characteristics. Those favorable characteristics (which apply as a mean and not on individual steps) result to a simple optimization rule, which is completely independent of the problem we need our algorithm to automatically solve. This way, we can pre-sort our database of already available solutions to save on processing time. Of course, we can try to optimize individual minimization steps, a more accurate but possibly more computationally-heavy process (it requires exact calculations for $p_{1,n}$ and $p_{2,n}$). In any case, we present a short series of powerful (albeit sometimes detrimental) theorems, which show global minimum existence and sufficient conditions to achieve optimality of the end-result. The simplest but most powerful of those conditions amounts to just having a large enough non-repeating database of known solutions. The database may be incomplete, but its large size ensures that the problem is approximated by the "best" solution, even if that solution is incomplete. In the end, we present a Final Algorithm which sums-up all necessary steps and practices for solving the problem of loose automated code generation.

Finally, we delve into more pragmatical discussions, examining ways to improve Database Import, essentially presenting an equivalue extension, for better recognizing human language.

Of course, the presented the theory is but a first step towards tackling a vast problem. Thus, numerous improvements are possible and further research can be conducted on almost all aspects overlapping with relevant fields. A rough approach towards recognizing this work's weaknesses and proposing possible solutions is available in the Further Study appendix.

A Further Study

A.1 Proof Machines

Instead of implementing a problem, our developed Final Algorithm can also be used as a proof machine, according to the following;

Proposition 9 (Tautology) *If $t_g(\mathcal{D}_n) \ominus \mathbb{Z} = \emptyset$, $\mathcal{D}_n$ is a tautology.*

Theorem 25 (Proof Machine) *If $\mathcal{H}_\infty \ominus \mathbb{Z} = \emptyset$ for the general algorithm for automated code generation, $\mathcal{H}$ is a tautology.*

Obviously, our methods donnot suffice for importing mathematical documents or libraries and new ones should be developed.

A.2 Equivalue Improvements

Research can be conducted on the exact equivalue definition and implementation. In particular, while still retaining equivalue properties, one could attempt to improve equivalue implementation and equivalue from lexicographic similarity, as well as equivalue extension rules.

The main issue to be addressed in equivalue implementation is to make plausible equivalue between expressions with different numbers of variables. For example, one could try criteria that take into account the multiplicity and ordering of variables within expressions. For example, it may be beneficial to consider the expressions $f(x_1, x_2, x_2, x_3)$ and $f(y_1, y_2, y_3, y_4)$ equivalue. A simple way to do this upon the current framework would be to generate temporary variables for each instance of multiplicity variables, while copying the original variable's comment context for those temporary variables. However, further study is needed to ensure only benefits from such functionality.

The shortcoming of equivalue from lexicographic similarity, or any bag-of-word approach for detecting similarity for that matter, is the inability to recognize context structure. Of course, thanks to flattening any recognized syntax structure will be fully analyzed into its core components. However, efficiently recognizing linguistic syntax lies outside the scope of this paper, where most text is assumed to be a concatenated bag-of-words. At most, our current approach consists of defining special predicates (such as *and* or the new line character) that crudely split problem descriptions into more formulas. Approaches aiming to cover this major deficiency (which results to a considerable loss of information due to effectively incomplete flattening) should also improve Comment Transformations for Variables to insert missing variables into already existing text in a context-wise manner.

Also, flattened formulas have maximum number of symbols $A+5$, where A is the maximum number of arguments a function or predicate within the examined application of Second-Order Logic. Therefore, the check $(c_1 \sim c_2) = true$ contains $\|c_1\| + \|c_2\| + 1 \leq 2A + 11$ symbols. For human language, we can use the conservative upper bound $A = 5$ (as humans cannot perceive more than 5 objects simultaneously). Thus, from Cook's complexity theorem, it follows that, in the ideal case of equivalue from lexicographic similarity completeness, we have a constant -albeit exceedingly large- upper bound for similarity checks. We can also see the imperativeness of enacting efficient flattening, as for expressions with higher 1-complexity Cook's complexity theorem will result to unreasonably large upper bounds.

Towards that end, for well-defined Second-Order Logic formulas (i.e. with a-priori defined predicates, functions and operations) we could use already existing algorithms to contextually analyze text. A class of especially promising algorithms is the generation of logical trees,[35–38] as their results can be translated to flattening in a detrimental way. To directly translate human-generated text to Second-Order Logic, one could also use syntax recognition techniques, which have strong foundations[39] and are constantly evolving.[40] Also, one could try textual recognition techniques used by translators[41] to contextually recognize predicates logically equivalent to *and* within their context.

A.3 Object-Oriented Code Generation

An important area of study is the extension of current techniques to encompass object-oriented code generation. Although the underlying method could be the same, there is a number of major problems to be addressed first.

First of all, methods should be developed to recognize class variables. On programing languages with strict data type definitions (such as C++ and Java), this does not pose a problem. However, doing so in higher-level programing languages (such as Python and LUA) is a challenging prospect. Still, a possible approach could through Attribute Grammar Trees,[55,56] which are already used by most object-oriented languages.

As mentioned before, variable and class detection is considerably straightforward for Python, as the object reference $THIS$ predicate (i.e. $self$) is always a function argument. Coupled with the ability to have multiple return values, this makes Python a promising field to apply our methods. However, our attempts are hampered a bit due to lose comment structure, encouraging a concentrated location of comments within the class's *help* function rather than being distributed before each class function declaration. In more strict programming languages, such as Java, comments are encouraged to have a favorable form, but it becomes imperative to explore the syntax tree for $THIS$ inference (Java is more favorable than C++ in this aspect, as functions can be declared and implemented only within their class).

Still, possible issues may arise from inheritance properties, which are not adequately represented in the current implementation. However, extending the equivalue implementation is a dynamic enough procedure that should be able to handle possible complications.

Finally, the prospect of generating complete classes could be explored. Towards that end, further research to detect possible class members should be conducted. Ideally, a cohesion metric, such as normalized cross-correlation between class member uses within the same function, could serve well as a maximization metric. This metric should be maximized alongside or after automatically generating class functions. In this approach, during code generation, function inputs and outputs would generate arguments or return statements only if not detected as class members.

A.4 Name-Based Context Matching

To further advance equivalue from lexicographic similarity, we could also introduce variable and function name matching to infer contextual similarity. Assuming that programmers use descriptive variable names, this way context can be generated even without relevant comments being available.

To successfully generate context from variables names, the latter should be split into individual words, according to either capitalization (for Camel notation - with consecutive capitalization denoting acronyms) or underscore existence (for Hungarian notation).

A naive approach would be to place detected words alongside (maybe next to) the variables they are generated from within the comments. However, to truly generate new context, new comments should be generated. Despite not being necessary, this attempt would be greatly improved by inferring contextual variable interactions and band multiple variables within the same new comment. This final prospect poses a number of challenges though, the most prominent of which being the contextual linking between words. and further research is needed to be integrated into developed methods.

B Alternate q-Complexity Definition

Throughout this work, we have used the proposed simple definition of q-complexity based one the q-Norm. However, one could be tempted to redefine it in ways that produce various favorable results. Such approaches will be discussed here, along with their strong and weak points. In the end, for the sake of clarity, it is desirable to stick with the simple, symmetric and intuitive definition we presented before, which handles in the same way quantifiers, equations and functions and predicates. Still, some approaches could use some of the alternate definitions proposed here, as they can produce considerably powerful results.

The first aspect of q-Complexity to reconsider, is its evaluation for equations. In particular, whilst maintaining the rest of the definition, we could define $\|a = b\|_q = \|\{\|a\|_q, \|b\|_q\}\|_q + 1 - \frac{3}{q}$. This definition maintains the notion of depth as-is (while $q \to \infty$, $\frac{3}{q} \to 0$), but has other implications on our analysis. On the up side, this way flattening maintains non-trivial information for 1-complexity; for our new definition $\|\mathcal{M} \ominus \mathbb{Z}\| = \|flattened(\mathcal{M}) \ominus \mathbb{Z}\|$. Also, we can perform syntactical checks, as a two-way implication with any model that contains a formula with negative complexity, shows an syntax error (as negative complexity can be achieved only by trying to assign the empty formula to itself or a variable). However, we distance ourselves from our close relation with entropy; theorems regarding entropy don't hold at all in this case. We can observe the advantages of using Second-Order Logic with Equation, as it already allows separate handling for equations.

Having produced a result with the previous definition, we may be tempted to alter it even further in an attempt to enforce 0 complexity value for the detrimental model. To do so, we may also re-define $\|^{\circ}x\, f\|_q = \|\{1, \|f\|_q\}\|_q + 1 - \frac{2}{q}$. Combining this with the previous re-definition we may actually produce the result $\|\mathbb{Z}\| = 0$. Actually, in the space of the combined definition, Algebrization retains complexity. We could use this definition for a more coherent approach, without even having to use the detrimental model, but we have opted for the more general analysis that outlines the reasoning and can be freely expanded upon.

Another interesting notion is to directly use entropy instead of 1-Complexity for our analysis. This approach is indeed plausible and yields similar results to 1-Complexity in simple cases, but has the major drawbacks of being computationally-intensive, not intuitive for defining model complexity as the sum of individual formula complexities (other definitions would not maintain the required linearity) and unable to represent structural complexity without flattening. In particular, the latter proves detrimental when experimenting with new approaches or using simpler methods, such as equivalue from lexicographic similarity. For example, the more complex structure "A(B(C(1),2, 3),4,5)" has entropy $\log_2 8$ and complexity 11, while the (structurally simpler) linear logic $A(1, 2, 3, 4, 5, 6, 7, 8)$ has entropy $\log_2 9$ and complexity 9.

In the previous example, it becomes apparent that we could alter q-Complexity to give more weight to function nesting. Thus, we may redefine $\|P(t_1, ..., t_n)\|_q = \|\{\|t_1\|_q + 1, ..., \|t_n\|_q + 1\}\|_q$. This definition maintains the notion of depth as-is, while giving more weight to arguments (even if the empty formula would be used as an argument). It also makes impossible the use of functions and predicates without arguments, as they would have 0 complexity and would thus be the empty formula. To cover this weakness, we could handle them as variables

(but this can get confusing when actually implementing produced code). Additionally, by straying from the close relation with entropy, its effectiveness for classification is uncharted territory.

Taking the above into account, if we allow ourselves to disregard relations with entropy, we can define q-Complexity in the following alternate way;

Definition 21 (Alternate q-Complexity)

- $\|f\|_q = 0 \Leftrightarrow f = \emptyset$
- $\|f\|_q = 1 \Leftrightarrow f \in \{variables\} \cup \{constants\}$
- $\|P(t_1, ..., t_n)\|_q = \big\|\{\|t_1\|_q + 1, ..., \|t_n\|_q + 1\}\big\|_q$ for $P \in \{functions\} \cup \{predicates\}$
- $\|a = b\|_q = \big\|\{\|a\|, \|b\|\}\big\|_q + 1 - \frac{3}{q}$
- $\|a \circ b\|_q = \big\|\{\|a\|, \|b\|\}\big\|_q + 1$ for $a, b \neq \emptyset$ and $\circ \in \{\leftrightarrow, \rightarrow, and, or\}$
- $\|^\circ x\, f\|_q = \|\{1, \|f\|_q\}\|_q + 1 - \frac{2}{q}$ where x is a variable and $^\circ \in \{\exists, \forall\}$

For the above definition we can show that;

Theorem 26 (Flattening Alternate q-Complexity) *For any model $\mathcal{M} \in \mathbb{SOL}^*$, using the alternate q-Complexity definition, we obtain that;*

$$0 \leq \|flattened(\mathcal{M})\|_q - \|\mathcal{M}\|_q \leq \big(3 - \tfrac{3}{q}\big)^{|\mathcal{M}|} \tag{33}$$

From this theorem, we can infer that Alternate 1-Complexity is retained through flattening.

References

1. Peter Bartalos, Maria Bielikova
   Automatrd Dynamic Web Service Composition: A Survey and Problem Formalization
   Computing and Informatics, Volume 30, pp. 793827, 2011
2. J. E. Haddad, M. Manouvrier and M. Rukoz
   TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition
   IEEE Transactions on Services Computing, vol. 3 (1), pp. 73-81, 2010
3. K. Verma, K. Gomadam, A.P. Sheth, J.A. Miller, Z. Wu
   METEOR-S Approach for Configuring and Executing Dynamic Web Processes
   Technical Report, 6-24-05
4. R. Milner
   Communication and concurrency
   Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989
5. R. Milner
   Communicating and mobile systems: the pi-calculus
   Cambridge University Press, Fifth edition, 2004
6. J. Rao, P. Kngas, M. Matskin
   Logic-based Web Service Composition: From Service Description to Process Model
   Proceedings, IEEE International Conference on Web Services, pp. 446-453, 2004
7. G. Baryannis, D. Plexousakis
   Automated Web Service Composition: State of the Art and Research Challenges
   Tech. Rep. TR-409, ICS-FORTH, Octomber 2010

8. Y. Hwang, E.P. Lim, C.H. Lee, C.H. Chen
Dynamic Web Service Selection for Reliable Web Service Composition
IEEE Transaction on Services Computing, Volume 1, no. 2, pp. 104-116, 2008
9. A. Brogi, S. Confini
Ontology- and behavior- aware discovery of web service compositions
International Journal on Cooperative Information Systems, 17(3): pp. 319-347, 2008
10. Q. Sheng, B. Benatallah, Z. Maamar, A. Ngu
Configurable Composition and Adaptive Provisioning of Web Services
IEEE Transaction on Services Computing, Volume 2, no. 1, pp. 3449, 2009
11. Mohammad Alrifai, Thomas Risse, Wolfgang Nejdl
A Hybrid Approach for Efficient Web Service Composition with End-to-End QoS Constraints
Journal; ACM Transactions on the Web (TWEB), Volume 6 Issue 2, Article No.7, May 2012
12. Chung-Wei Hang, Munindar P. Singh
Trustworthy Service Selection and Composition
Journal; ACM Transactions on Autonomous and Adaptive Systems, Volume 6, Issue 1, Article No. 5, February 2011
13. John R. Koza
Genetic Programming IV: Routine Human Competitive Machine Intelligence
2003
14. Miro Samek
Turning automatic code generation upside down
February 14th, 2012
15. Ricardo Aler Mur
Automatic Inductive Programming
Tutorial at the International Conference on Machine Learning, Carnegie Mellon, Pittsburgh, Pennsylvania, U.S.A., 2006
16. Dirk Ourston, Raymond J. Mooney
Theory refinement combining analytical and empirical methods
Artificial Intelligence 66, pp. 273-309, 1994
17. Stephen Muggleton, Michael Bain, Jean Hayes-Michie, Donald Michie
An Experimental Comparison of Human and Machine Learning Formalisms
18. David E. Goldberg, Jerry S. Dobrovolny
Scalable Optimization via Probabilistic Modeling; from Algorithms to Applications
Springer, May 2006
19. Kumara Sastry, David E. Goldberg
Probabilistic Model Building and Competent Genetic Programming
Genetic Programming Series Volume 6, pp. 205-220, 2003
20. J. Vaananen
Second-Order Logic and Foundations of Mathematics
Bulletin of Symbolic Logic 7, pp. 504-520, 2001
21. R. Fagin
Generalized First-Order Spectra and Polynomial-Time Recognizable Sets
Complexity of Computation, ed. R. Karp, SIAM-AMS Proceedings 7, pp. 2741, 1974
22. Comen, Leiserson, Rivest, Stein
Introduction to Algorithms
3rd edition, 2009
23. Raymond M. Smullyan
First-Order Logic
1995
24. Jean-Franois Couchot, Thierry Hubert
A Graph-based Strategy for the Selection of Hypotheses
6th International Workshop on First-Order Theorem Proving, 2007
25. Jean-Francois Couchot, A. Giorgetti, Nicolas Stouls
Graph-based Reduction of Program Verification Conditions
Automated Formal Methods '09, pp. 40-47, 2009

26. Ian Hodkinsona, Frank Wolterb, Michael Zakharyaschevc
Decidable fragments of first-order temporal logics
Annals of Pure and Applied Logic, Volume 106, Issues 1-3, pp. 85-134, 1 December 2000
27. David Deharbe, Silvio Ranise
Satisfiability Solving for Software Verification
International Journal on Software Tools for Technology Transfer, Volume 11, Issue 3, pp. 255-260, July 2009
28. Nils J. Nilsson, Richard E. Fikes
STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving
1970
29. Fabrice Nahon, Claude Kirchner, Helene Kirchner
Inductive Proof Search Modulo
Annals of Mathematics and Artificial Intelligence, Volume 55, Issue 1-2, pp. 123-154, February 2009
30. Steven M. Pincus
Approximate Entropy as a Measure of System Complexity
Proceedings of the National Academy of Sciences of the United States of America, Vol. 88, No. 6, pp. 22972301, 1990
31. Hirotogu Akaike
Information Theory and an Extension of the Maximum Likelihood Principle
Springer Series in Statistics, pp. 199-213, 1998
32. E. T. Jaynes
Information Theory and Statistical Mechanics
Physical Review, Volume 106, No. 4, 1957
33. George J. Klir
Uncertainty and Information; Foundations of Generalized Information Theory
34. Stephen A. Cook
The Complexity of Theorem-Proving Procedures
35. Rainer Koschke, Raimar Falke, Pierre Frenzel
Clone Detection Using Abstract Syntax Suffix Trees
36. Colin Howson
Logic with Trees
1997
37. http://gablem.com/logic
Truth Tree Solvers
38. Melvin Fitting
First-Order Logic and Automated Theorem Proving
Second Edition
39. W.A.Woods
Transition Network Grammars for Natural Language Analysis
1970
40. Alessandro Moschitti
Making Tree Kernels practical for Natural Language Learning
2006
41. Libin Shen, Jinxi Xu, Ralph Weischdel
A New String-to-Dependency Machine Translation Algorithm
42. Tao Xie, Suresh Thummalepenta, David Lo, Chao Liu
Data Mining for Software Engineering
2009
43. A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, D. B. Rubin
Bayesian data analysis
CRC Press, 2013
44. K. P. Murphy
Machine Learning: a Probabilistic Perspective *MIT Press, 2012*
45. N. L. Hjort, C. Holmes, P. Muller, S.G. Walke
Bayesian Nonparametrics
Number 28. Cambridge University Press, 2010

46. C. K. Williams, C. E. Rasmussen
Gaussian Processes for Machine Learning
2006
47. Mining Idioms from Source Code
Miltiadis Allamanis, Charles Sutton
School of Informatics, University of Edinburgh
48. Chris J. Maddison, Daniel Tarlow
Structured Generative Models of Natural Source Code
arXiv:1401.0514v2 [cs.PL], 20 Jun 2014
49. Paolo Giudici
Improving Markov chain Monte Carlo search for Data Mining *Article in Machine Learning, January 2003*
50. Ben Swanson, Eugene Charniak
Native Language Detection with Tree Substitution Grammars
50th Annual Meeting of the Association for Computational Linguistics, Proceedings, Volume 2, 2012
51. Matt Post, Daniel Gildea
Bayesian Learning of a Tree Substitution Grammar
Conference of the Association for Computational Linguistics-IJCNLP, Proceedings, 2009
52. Yee Whye Teh, Michael I. Jordan
Hierarchical Bayesian Nonparametric Models with Applications
2009
53. Celine Rouveirol
Flattening and Saturation: Two Representation Changes for Generalization
Machine Learning, Volume 14, pp. 219-232, 1994
54. Ryszard S. Michalski
Pattern Recognition as Rule-Guided Inductive Inference
IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume PAM 1-2, No. 4, July 1980
55. Kenneth C. Louden
Compiler Construction: Principle and Practice
56. William M. Waite, Gerhard Goos
Compiler Construction
1996
57. Gerard Berry, Ravi Sethi
From Regular Expressions to Deterministic Automata
Theoretical Computer Science, Volume 48, pp. 117126, 1986
58. Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, Randy H. Katz
Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection
Technical Report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006
59. Michael Gelfond, Vladimir Lifschitz
The Stable Model Semantics for Logic Programming
60. Fangzhen Lin, Yuting Zhao
ASSAT: computing answer sets of a logic program by SAT solvers
Artificial Intelligence, Volume 157, Issues 12, pp. 115137, August 2004