Towards Modal Software Engineering

Ramy Shahin
University of Toronto, Canada
rshahin@cs.toronto.edu

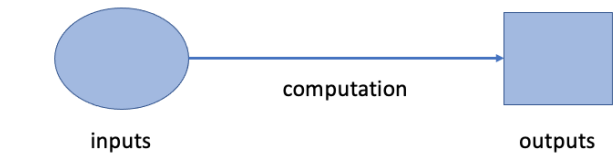
***Abstract*—In this paper we introduce the notion of Modal Software Engineering: automatically turning sequential, deterministic programs into semantically equivalent programs efficiently operating on inputs coming from multiple intersecting worlds. We are drawing an analogy between modal logics, and software application domains where multiple sets of inputs (multiple worlds) need to be processed efficiently. Typically those sets highly overlap, so processing them independently would result in a lot of redundancy, resulting in lower performance, and in many cases intractability. Three application domains are presented: reasoning about feature-based variability of Software Product Lines (SPLs), probabilistic programming, and approximate programming.**

I. Introduction

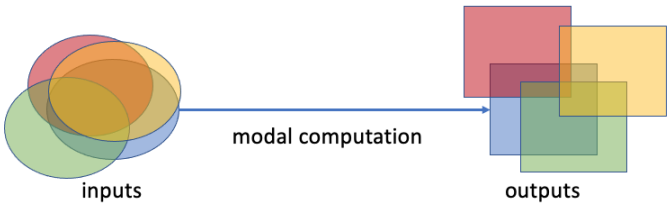
Computer programs take concrete values as inputs, perform a sequence of operations on those inputs (assuming sequential, deterministic programs), and generate concrete outputs. All values involved in this process (inputs, intermediate values, and outputs) can be considered an abstraction of the *world* in which the program operates. A single world is typically assumed by a program. For example, a program variable `v` of type `T` maps to a single value of that type at any point in time, and a conditional expression takes a single branch based on what its guard evaluates to.

In the realm of mathematical logic, reasoning about multiple worlds at the same time has been the motivation behind *modal logics*. For example, temporal logics reason about not only the world at the current instance of time, but also about different instances (worlds) in the future. The *S5* logic [1] is another example, distinguishing *essential* truth (true in all worlds) and *possible* truth (true only in some worlds). Modal logics have many applications in numerous domains, including verification of concurrent systems [2], reasoning about computer programs [3], artificial intelligence [4], robotics [5], linguistics [6], and philosophy [7]. This is a strong indication of how complex real-life problems are, and that new logical formalisms are continuously being introduced to address them effectively.

In several practical domains, software systems are also required to compute over inputs coming from multiple, potentially overlapping, worlds. Values varying from one world to another are appropriately labeled to ensure that outputs are also labeled accordingly. For example, a search engine storing multiple versions of a database has to effectively index and query multiple logical databases, each considered a world. Physically though, since those versions are only slightly different from one another, it is highly inefficient to store them and perform queries over them separately. Mining Software Repositories (MSR) [8] is an example of a domain where such highly overlapping databases (successive versions of software artifacts) are common, and the scalability of different MSR systems is considered a challenge both to researchers and practitioners because of that [9].



(a) A computation maps inputs to outputs.



(b) A modal computation maps inputs from multiple overlapping worlds, to potentially overlapping outputs corresponding to those worlds.

Fig. 1: Computation vs. modal computation.

In this paper we follow the analogy of modal logics, and argue that computing over multiple worlds is common around us, and needs to be addressed systematically (Sec. II). The high-level logic of a program operating over multiple worlds is essentially the same as that of a single-world program. At the implementation level though, it needs to compute over multi-valued variables, where each value is appropriately labeled by an identifier of the world(s) it belongs to. The implementation also needs to be *efficient*, i.e., with as few redundant computations as possible. We also argue that since the high-level program logic is oblivious to the number of worlds it is operating on, turning a single-world program into a modal (multi-world) one can be performed automatically, in a way similar to compiler optimizations. We provide a vision for rewriting single-world programs into their efficient modal counterparts in Sec. III.

Fig. 1 illustrates the difference between single-world computations (Fig. 1a), mapping a single set of inputs to a single set of outputs, and modal computations (Fig. 1b), working on multiple potentially overlapping sets of inputs at the same time and generating outputs for each world, also potentially overlapping. There are cases where each world is explicitly

labeled, and others in which they can be only quantitatively described. Examples of each are discussed in Sec. IV.

II. Motivation

This work has been originally motivated by the analysis of *Software Product Lines (SPLs)*. An SPL is a family of related software products, developed together from a common set of artifacts [10]. The unit of variability in an SPL is a *feature*. Each feature can be either present or absent in each of the product variants of the SPL. Given this combinatorial nature of features, the number of products grows exponentially with the number of features. Each product is defined by a *feature configuration*, which is the set of features present in that product. A *feature expression* is a propositional formula over features, denoting a set of products.

Efficiently analyzing SPLs has been an active area of software engineering and programming languages research for more than a decade. Software analyses (of source code, transition system models, graphical models, etc...) typically work on a single product at a time, and they need to be modified to be applicable to a whole SPL. This is an instance of the problem of taking a single-world program, and turning it into a semantically equivalent multi-world one.

The product line example in Listing 1 has two features: FA and FB. Assuming all product combinations are valid, four products can be generated from this example with the following feature combinations: {}, {FA}, {FB}, and {FA, FB}. Assume we have a static program analysis for detecting division-by-zero errors. Only one of the four configurations ({FB}) has a statically detectable division-by-zero error, where the value of variable `c` is not incremented in line 5 because FA is absent, and line 9 (specific to feature FB) divides `(x + y)` by `c`. The original single-world analysis can be applied to each of the four products of this SPL individually, reporting results for each. However, real-life SPLs usually have hundreds, sometimes thousands, of features (The Linux kernel has more than 10,000 features [11]), so enumerating all products and analyzing them one-by-one is not generally feasible. In addition, different product variants have a lot in common, and leveraging this amount of commonality can make analyzing the whole product line at once much more efficient than analyzing the products one at a time.

Several analyses of different kinds have been manually redesigned and re-implemented to support variability [12], [13], [14], [15], [16], [17]. Re-implementing an analysis is a lengthy and error-prone process, so several attempts have been made to come up with systematic or automated approaches to this process. Examples include systematic variability-aware abstract interpretation [18], SPL data-flow analyses based on the IFDS framework [19], [20], variability-aware analyses written in Datalog [21], and program analyses written in a functional language [22].

We argue that the research effort that has been put in SPL analysis (also known as *variability-aware lifting*) can be generalized to domains other than SPLs, where efficient computations are needed over multiple overlapping worlds.

```
1  int foo(int x, int y)
2  {
3      int c = 0;
4  #ifdef FA
5      c++;
6  #endif
7
8  #ifdef FB
9      return (x + y) / c;
10 #else
11     return (x + c) / y;
12 #endif
13 }
```

Listing 1: Example of a Software Product Line (SPL).

In SPL analysis, each product is considered a world, labeled by a feature configuration. Inputs, outputs and intermediate values of the analysis are labeled by propositional formulas over features (known as *Presence Conditions (PCs)*) denoting sets of products. Other problem domains can be mapped into this same formulation of worlds.

There are also cases where the individual worlds are not explicit, but rather quantified. For example, in a *probabilistic program*, a variable `v` can have the value `{(7,0.2), (9,0.8)}`, i.e., the value `7` with probability 0.2 (20% of possible worlds), and the value 9 with probability 0.8 (80% of possible worlds). Another example is *approximate computing*, where instead of a concrete value, a variable can at any point in time be defined by a minimum-maximum interval. For example, `v = [4..9]` means v is approximated to the range 4 to 9 inclusively.

III. Vision

Mathematical logic provides a formal framework for reasoning about truth. There are many cases though where truth and falsehood are not absolute; for example a statement can be true now but not in the future, or it can be true from one person's perspective and false from another's. Modal logics [23], [24] add qualifiers to logical formulas, and those qualifiers denote modalities, or modes of truth. For example, in the modal logic of necessity and possibility *S5* [1], the $\Box$ connective indicates necessity (truth in all possible worlds), while the $\Diamond$ connective indicates possibility (truth in some possible world(s)). Those are two unary connectives that qualify the truth of propositional formulas.

Given how logic and computation are sometimes viewed as two sides of one coin (e.g., the Curry-Howard isomorphism [25]), computing, rather than just reasoning, about multiple worlds also has potentially many applications. Examples include explicit variability in Software Product Lines (SPLs), and quantitative variability in probabilistic models.

Our goal is to take a single-world program, and automatically convert it into a semantically equivalent multiple-world (or modal) program. Modalities labeling the individual worlds

x = {(−7,FA),(3, $\neg FA$)}
y = {(1, $FA \wedge FB$), (8,$FA \wedge \neg FB$), (4, $\neg FA \wedge FB$), (10, $\neg FA \wedge \neg FB$)}
z = {(5, True)}

(a) Arguments x, y, and z.

```
int foo(int x, int y, int z) {
    return bar(x, y) + baz(z);
}
```

(b) Function foo.

x	y	z	***LABEL***
-7	1	5	$FA \wedge FB$
-7	8	5	$FA \wedge \neg FB$
~~-7~~	~~4~~	~~5~~	~~$FA \wedge \neg FA$~~
~~-7~~	~~10~~	~~5~~	~~$FA \wedge \neg FA$~~
~~3~~	~~1~~	~~5~~	~~$\neg FA \wedge FA$~~
~~3~~	~~8~~	~~5~~	~~$\neg FA \wedge FA$~~
3	4	5	$\neg FA \wedge FB$
3	10	5	$\neg FA \wedge \neg FB$

(c) Input vectors for foo.

Fig. 2: Variability-aware application of function foo to modal arguments x, y, and z, tracing the cross-product of modal arguments (adapted from [22]).

or quantifying them, and the semantics of those modalities, are orthogonal to the rewriting process. We refer to the process of modal program rewriting as *modal lifting*.

For example, function foo in Fig. 2b takes three parameters of type int. A semantically equivalent lifted function would take three arguments of type $\texttt{int}^{\uparrow}$ instead, where values of type $\texttt{int}^{\uparrow}$ are sets of (int, LABEL) pairs. Variables x, y, and z in Fig. 2a are of type $\texttt{int}^{\uparrow}$, where labels are propositional formulas identifying sets of products in a software product line with two features FA and FB.

Each label within a modal value denotes a set of worlds, and has to satisfy two conditions: (1) *Disjointness*: within the same modal value, the sets of worlds denoted by the different labels have to be mutually disjoint. If they are not, then we would be allowing different values within the same world, which would result in non-deterministic evaluation of programs. (2) *Totality*: within the same modal value, the union of all the sets of worlds denoted by the different labels has to be equal to the set of all worlds. This ensures that each modal value is defined in each world.

In the case of software product lines, disjointness and totality on feature expressions are defined in [22]. Disjointness is defined as the propositional unsatisfiability of the conjunction of any pair of different labels within the same modal value. For example, the variable y in Fig. 2a has four labels: $FA \wedge FB$, $FA \wedge \neg FB$, $\neg FA \wedge FB$, and $\neg FA \wedge \neg FB$. Conjoining any two of them results in an unsatisfiable propositional formula. Totality on the other hand is defined as the disjunction of labels within a modal value resulting in a tautology. Again, this is the case for each of the variables in Fig. 2a.

In the context of software product lines, two automatic program lifting approaches have been proposed in [22]: *shallow lifting* and *deep lifting*. Shallow lifting takes a program as a black-box and generates the cross-product of program arguments, passing each combination of arguments to the original single-world program. For example, tuples of the cross-product of arguments in Fig. 2a are listed in Fig. 2c.

Algorithm 1 outlines how an n-ary function f is shallow lifted and applied to modal arguments $arg_1, ..., arg_n$. First, we calculate the cross product of the arguments (each of which is a set of pairs). Each element in the cross-product is an n-tuple of $(v, label)$ pairs. Within each element, if the intersection of the sets of worlds denoted by the labels is the empty set, then we can safely ignore this element. However, if the intersection is not empty, we apply the function f to the arguments $v_1, ..., v_n$. and add the result to the output set, which is returned at the end.

Input: f, arg_1, ..., arg_n
Result: applying shallow lifting of f to $args$
$output = \{\}$;
foreach
$((v_1, label_1), ..., (v_n, label_n)) \in (arg_1 \times ... \times arg_n)$
do
  if $(\bigcap_{i=1}^{n} label_i \neq \phi)$ **then**
    $output = output \cup \{f(v_1, ..., v_n)\}$;
  end
end
return $output$;

Algorithm 1: Shallow lifting algorithm.

Applying the shallow lifting algorithm to the example in Fig. 2, the first step is calculating the cross-product of arguments x, y, and z (Fig. 2c). For each row in the table, *LABEL* is the conjunction of presence conditions of the components of x, y, and z in this row. Recall that conjunction is the logical operator corresponding to intersection of sets denoted by propositional formulas (feature expressions). Rows crossed out have unsatisfiable labels, i.e., they correspond to the empty set of worlds, so they are ignored, leaving out four input vectors.

Shallow lifting suffers from the inability to exploit commonalities and eliminate redundancies, and thus can be further optimized. For example, function foo in Fig. 2b internally passes x and y to bar, and passes z to baz. Assuming the values in Fig. 2a, z happens to be the constant value 5 across all configurations, so shallow-lifted foo ends up calling baz four times, passing the same argument each time. Shallow lifting treats foo as a black-box, so the opportunity to eliminate the redundant calls is not visible.

Deep lifting on the other hand inspects the internals of a

	Software Product Lines	Probabilistic Programming	Approximate Programming
Label Representation	Propositional formula	Probability $\in [0..1]$	MIN and MAX
Cardinality	> 0	> 0	2
Intersection	conjunction	multiplication	Group by MIN, MAX
Union	disjunction	addition	Group by MIN, MAX
Emptiness	unsatisfiability	0.0	$v_{MAX} < v_{MIN}$
Disjointness	$\forall i \neq j \cdot \texttt{unsat}(l_i \wedge l_j)$	$\Sigma_i l_i \leq 1.0$	1 MIN and 1 MAX
Totality	$\bigvee_i l_i = \texttt{True}$	$\Sigma_i l_i = 1.0$	1 MIN and 1 MAX

TABLE I: Comparing modalities, their invariants, and operations across three application domains: feature-based reasoning of software product lines, probabilistic programming, and approximate programming.

program, rewriting it to maximize sharing of common values, and to minimize redundancy. The idea behind deep lifting is to statically push the calculation of cross-products of modal values as deep as possible down the call tree of a program. For example, `foo` is re-written into `foo`$^{\uparrow}$ as follows:

```
int↑ foo↑ (int↑ x, int↑ y, int↑ z) {
    return
        shallowLift((+),
                    shallowLift(bar, x, y),
                    shallowLift(baz, z));
}
```

Function `foo` internally calls three functions/operators: `bar`, `baz`, and the (+) operator. Each of the three can be shallow lifted using Algorithm 1, and their shallow lifted counterparts operate on `int`$^{\uparrow}$ rather than `int` arguments. This way, the expression `shallowLift(baz, z)` within `foo`$^{\uparrow}$ calls the underlying function `baz` on the single component of argument `z` only once. If any of `bar` or `baz` is not a primitive function, it can be similarly rewritten into a deep-lifted function, which is then called from within `foo` instead of using `shallowLift`. This process can recursively applied down the call tree, rewriting the whole program. Deep lifting rewrite rules for a subset of Haskell including conditional expressions, pattern matching, and polymorphic lists have been presented in [22] in the context of lifting program analyses to software product lines.

IV. Applications

In this section we show how modal rewriting is oblivious to the specific modality used, which means the same lifting approach can be used across different kinds of modalities. We compare three application domains: Featured-based variability of software product lines, probabilistic programming, and approximate programming. The modalities of each application domain, their invariants and operators are summarized in Table I.

A. Feature-based Variability

When reasoning about Software Product Lines (SPLs), each feature is represented as a propositional symbol, and a set of products is denoted by propositional formulas over features (usually referred to as *feature expressions*). The set of worlds in this context is the set of products, and labels within modal values are feature expressions.

A modal feature-based value has at least one element (cardinality > 0). Intersection of sets of products denoted by feature expressions is conjunction of feature expressions. Similarly, union is disjunction. A set denoted by a feature expression is empty when the feature expression is an unsatisfiable formula.

Disjointness of feature expressions is the same as set disjointness: sets are disjoint when their pair-wise intersection is empty. Translating this to feature expressions, a collection of feature expressions is disjoint if their pair-wise conjunction is unsatisfiable. Similarly, totality of a collection of sets is the universe being equal to the union of that collection. In propositional logic, the tautology `True` is analogous to the universe of all worlds, and disjunction is analogous to set union.

B. Probabilistic Programming

Probabilistic programming surprisingly has a lot in common with feature-based variability. The main difference is that feature expressions define explicit sets, while probabilities quantify over them. For example, a probability of 0.8 indicates 80% of all worlds, without naming which 80%. This kind of quantitative description of sets has many applications, particularly in modeling and simulation.

Labels for probabilistic modal values come from the range [0.0..1.0]. Cardinality has to be greater than zero, same as in feature-based modalities. Assuming probabilistic independence, intersection of two sets denoted by probabilities is the multiplication of those probabilities. Similarly, union is their addition, and a probability of 0.0 denotes the empty set.

Because probabilities do not define explicit sets, disjointness is also quantitative rather than explicit. A necessary condition (but not always adequate) for disjointness of a collection of sets denoted by probabilities is that the sum of those probabilities is at most 1.0. Totality makes this bound more strict though, turning the inequality into an equality.

C. Approximate Programming

Approximate programming has many applications, particularly in software systems running in power-saving mode [27], and systems reading inputs from imprecise sensors. Approximation modalities can be represented in many ways, including a fidelity of a single value, or a range of possible values (minimum and maximum) over a continuous domain. We follow the latter representation in this example.

Cardinality here is fixed at two, since each modal range value will have exactly two sub-values: one labeled as MIN

and the other labeled as MAX. Values in this MIN..MAX range form a uniform distribution of possible values. Intersection and union are again different in this case because they both boil down to grouping MIN values together and MAX values together. A range is empty when the value labeled by MAX is less than that labeled by MIN.

Similarly, the disjointness and totality invariants are enforced by the fact that there is exactly one MIN value and exactly one MAX value. Lifted programs will compute results for the MIN value and MAX value, labeling the results. accordingly.

V. Conclusion and Future Work

In this paper we introduced the notion of Modal Software Engineering: automatically turning sequential, deterministic programs into semantically equivalent programs, that efficiently operate on inputs coming from multiple intersecting worlds. We drew an analogy between modal logics, and software application domains where multiple sets of potentially overlapping inputs (multiple worlds) need to be processed efficiently. Processing each world independently instead would result in a lot of redundancy, resulting in lower performance, and in many cases intractability. Three application domains have been presented: reasoning about feature-based variability of Software Product Lines (SPLs), probabilistic programming, and approximate programming.

For future work, we plan to investigate the three example domains presented in this paper in more depth, identifying domain-specific problems and reflecting on whether they are related to problems in other domains. Examples of such problems are dependent variables in probabilistic programs, and approximate computing over categorical values. An orthogonal research direction we plan to pursue is studying the different program rewriting techniques (and their trade-offs) for lifting programs into modal ones.

Acknowledgments

The author thanks the anonymous reviewers for their feedback and insightful suggestions.

References

[1] B. F. Chellas, *Modal Logic: An Introduction*. Cambridge University Press, 1980.

[2] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of Model Checking*, 1st ed. Springer Publishing Company, Incorporated, 2018.

[3] V. R. Pratt, "Application of modal logic to programming," *Studia Logica: An International Journal for Symbolic Logic*, vol. 39, no. 2/3, pp. 257–274, 1980. [Online]. Available: http://www.jstor.org/stable/20014985

[4] D. McDermott and J. Doyle, "Non-monotonic logic i," *Artificial Intelligence*, vol. 13, no. 1, pp. 41 – 72, 1980, special Issue on Non-Monotonic Logic. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370280900120

[5] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas, "Temporal logic motion planning for mobile robots," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005, pp. 2020–2025.

[6] L. Moss and H. Tiede, "19 applications of modal logic in linguistics," *Studies in Logic and Practical Reasoning*, vol. 3, pp. 1031–1076, 12 2007.

[7] L. Humberstone, *Philosophical Applications of Modal Logic*, ser. Studies in Logic. College Publications, 2016. [Online]. Available: https://books.google.com.eg/books?id=Wi8PkAEACAAJ

[8] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of software maintenance and evolution: Research and practice*, vol. 19, no. 2, pp. 77–131, 2007.

[9] W. Shang, B. Adams, and A. E. Hassan, "An experience report on scaling tools for mining software repositories using mapreduce," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 275–284. [Online]. Available: https://doi.org/10.1145/1858996.1859050

[10] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[11] S. Nadi and R. Holt, "The Linux Kernel: A Case Study of Build System Variability," *J. Softw. Evol. Process*, vol. 26, no. 8, p. 730–746, Aug. 2014.

[12] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation," in *Proc. of OOPSLA'11*. ACM, 2011, pp. 805–824.

[13] P. Gazzillo and R. Grimm, "SuperC: Parsing All of C by Taming the Preprocessor," in *Proc. of PLDI'12*. ACM, 2012, pp. 323–334.

[14] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1069–1089, Aug. 2013.

[15] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A Classification and Survey of Analysis Strategies for Software Product Lines," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014.

[16] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik, "Lifting Model Transformations to Product Lines," in *Proc. of ICSE'14*. New York, NY, USA: ACM, 2014, pp. 117–128.

[17] A. Von Rhein, J. Liebig, A. Janker, C. Kästner, and S. Apel, "Variability-Aware Static Analysis at Scale: An Empirical Study," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, Nov. 2018.

[18] J. Midtgaard, A. S. Dimovski, C. Brabrand, and A. Wąsowski, "Systematic Derivation of Correct Variability-aware Program Analyses," *Sci. Comput. Program.*, vol. 105, no. C, pp. 145–170, Jul. 2015.

[19] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proc. of POPL'95*. ACM, 1995, pp. 49–61.

[20] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini, "SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years," in *Proc. of PLDI'13*. ACM, 2013, pp. 355–364.

[21] R. Shahin, M. Chechik, and R. Salay, "Lifting Datalog-based Analyses to Software Product Lines," in *Proc. of ESEC/FSE'19*. New York, NY, USA: ACM, 2019, pp. 39–49.

[22] R. Shahin and M. Chechik, "Automatic and Efficient Variability-Aware Lifting of Functional Programs," in *Proc. of OOPSLA'20*, 2020.

[23] P. Blackburn, M. d. Rijke, and Y. Venema, *Modal Logic*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[24] M. Huth and M. Ryan, *Logic in Computer Science (2nd ed.)*. Cambridge University Press, 2004.

[25] R. Nederpelt and H. Geuvers, *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[26] R. Shahin and M. Chechik, "Variability-aware datalog," in *Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings*, ser. Lecture Notes in Computer Science, E. Komendantskaya and Y. A. Liu, Eds., vol. 12007. Springer, 2020, pp. 213–221. [Online]. Available: https://doi.org/10.1007/978-3-030-39197-3_14

[27] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2893356