Learning Latent Memory Models from Litmus Tests*
(Technical Report)

XUANKANG LIN, Purdue University

In real world, both hardware architectures and programming languages relax their concurrent semantics to exploit full concurrency power. Such relaxed environments are inevitably error-prone for programmers. Researchers have been devising memory consistency models (MCM) to regulate what weak memory behaviors are allowed, thus granting various optimizations under the hood. Such formal model specifications can be used to reason about program correctness. However, they are cumbersome and limit programmers to those a few already formalized environments. More fine-grained efforts regarding only some subpart of the specification are potentially prohibited.

Meanwhile, litmus tests have been created to informally express the key ideas behind each model. They are extremely useful in testing and validating both specifications and architecture, language implementations. Litmus tests are intentionally made to be tiny programs, hence they are much more programmer-friendly comparing to complex formal specifications. Litmus tests are easily accessible and are more modular comparing to model specifications.

In this paper, we present a novel approach that can learn the latent memory model from a collection of litmus tests. The output model is able to predict whether a new execution conforms to the weak memory behaviors exhibited in those input tests. Few domain knowledge is required, making it well apply to both hardware and language models including TSO, PowerPC, ARM, and part of C11. Preliminary experiments have shown that the accuracy and F1 score of our learned model can be as high as 95%+.

CCS Concepts: •**Software and its engineering** → **General programming languages;** •**Social and professional topics** → *History of programming languages;*

Additional Key Words and Phrases: Memory Consistency Models, Specifications, Litmus Tests, Conditional Random Fields, Decision Tree

ACM Reference format:
Xuankang Lin. 2017. Learning Latent Memory Models from Litmus Tests. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 20 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

To exploit full concurrency power, both hardware architectures (Flur et al. 2016; Sarkar et al. 2011; Sewell et al. 2010) and programming languages (Batty et al. 2016, 2011; Manson et al. 2005) have devised memory consistency models (MCM) to regulate what weak memory behaviors are possible to happen, thus granting various optimizations under the hood. With greater flexibility, the relaxed environments make it even harder for programmers to reason about concurrent programs. Bugs have been found on PostgreSQL due to PowerPC weak memory behaviors (Alglave et al. 2013), in Java library implementations (Dice 2009), and may even lead to 12 million loss (12M 2013).

Formalized memory model specifications are designed to allow programmers to reason about their concurrent programs. However, such specifications are usually very sophisticated, they are neither easy to design nor to

*This informal report is done by Xuankang Lin with the help from Xiao Zhang, and supervision from Prof. Suresh Jagannathan. The author is to blame for any inconsistencies, confusion, incompleteness, or plain nonsense in the report.

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

comprehend. It is also possible that several years after a new model is established, some internal defects are reported, e.g. some common compiler optimizations are found unsound (Sevcík and Aspinall 2008; Vafeiadis et al. 2015). The cumbersome specifications limit programmers to those a few environments which have been formalized. More fine-grained efforts regarding only some subpart of the specification are potentially prohibited.

Meanwhile, litmus tests have been created to informally express the key ideas behind each model. These tiny pieces are extremely useful in conveying the specifications (Maranget et al. 2012) and in testing of architecture implementations (Alglave et al. 2015, 2014). Such litmus tests are much more programmer-friendly comparing to complex formal specifications, they are intentionally designed to be small, so as to be well understood. Multiple litmus tests collectively depict one memory model. It provides a way for programmers to decompose stunting memory model specifications into subparts that are truly relevant to their application.

It is simpler to understand if such behavior exists (or never appears) by a less than 10 lines of code, while expressing and understanding how to allow (or forbid) such behavior in a formal specification requires much heavier efforts.

In this work, we present a novel approach that can learn the latent memory model from a collection of litmus tests. The output model is able to predict whether a new execution conforms to the weak memory behaviors exhibited in those input tests. We require as few domain knowledge as possible so as to generalize to both hardware models and language models. Our approach is based on two key insights:

(1) We notice that different memory models manifest different degrees of weak memory behaviors. Hence, we feed litmus tests to a "weakest" memory model to expose all sorts of weak memory behaviors. Then well-defined litmus conditions can be used to label different concrete executions according to condition. Eventually positive executions are supposed to exhibit weak memory behaviors allowed by the latent model, and negative executions are exhibiting at least one weak memory behavior that is forbidden by the latent memory model. The intuition is that by using a weakest model, all weak memory behaviors should have been exposed and can be used by our learning framework to learn a classifier that approximates the actual latent memory model.
(2) The Domain Specific Languages (DSL) used to define different memory models share a lot in common. A common pattern in formalizing axiomatic memory models is to first identify some "basic relations", and then use a finite set of composition operators to define "derived relations" with domain knowledge. Finally, axioms regarding all these relations are devised to regulate the allowed and forbidden behaviors. We notice that almost all axioms are talking about cyclic properties and these global structural properties can be approximated by local neighboring relation pairs together with features regarding event pairs.

We have implemented the approach and conducted experiments on TSO (Alglave et al. 2013), PowerPC (Alglave et al. 2013, 2010; Sarkar et al. 2011), ARM (Sarkar et al. 2011), part of C11 (Batty et al. 2016; Vafeiadis et al. 2015) datasets. Preliminary experiments have shown that the accuracy and F1 score of our learned model can be as high as 95%+.

The rest of the paper is organized as follows: $\xi 2$ uses a concrete example to better explain the entire workflow of our approach; $\xi 3$ describes the forms of litmus tests that we take as input and memory models that our outputs try to approximate. The overall approach is elaborated in detail in $\xi 4$. Our Decision Tree based learning framework is explained in $\xi 5$, while our Conditional Random Field based learning framework is explained in $\xi 6$. Statistics of experiments are listed in $\xi 7$ to demonstrate the feasibility of our approach. Related works are discussed in $\xi 8$ and finally the conclusion and future work are summarized in $\xi 9$.

1.1 Discussion

In this report, more formal stories and techniques will be presented in normal sections. To make it clear, those ideas, various design choices, or failed attempts will be discussed at the end of each section. Those raw data, or raw information that eventually turns out to be irrelevant to the project will not appear in this report.

Throughout the project, there exists one fundamental discrepancy between the non-100% guarantee provided by machine learning based techniques and the 100% soundness expectation of a learned specification which undermines the overall story. One possible solution is to add a verifier to the workflow and redo the prediction if it fails to be correct in a task such as test generation.

2 MOTIVATING EXAMPLE

3 PRELIMINARIES

In this section, we briefly introduce the input litmus tests and the ideal memory consistency model that our output tries to approximate.

3.1 Litmus Tests

The input litmus tests (P, C) are essentially small piece of program P with a final condition C that asserts the final states of all executions. The litmus condition C is usually quantified by $\exists$ or $\forall$ operator.

In this paper, we follow the syntax of litmus tests used in herd7 toolkit (Alglave et al. 2014). Quantifiers such as $\exists$, $\neg\exists$, $\forall$ are used to describe the final states.

Most litmus tests we found are using formulas of the shape $\exists$ or $\neg\exists$. They are used to specify whether one particular weak memory behavior can be observed. In this work, however, litmus tests are supposed to list all acceptable final states so that those behaviors not covered by the condtion can be treated as forbidden. Later on, we can use the litmus condition to label generated concrete executions and learn the latent model which results in this allowed/forbidden difference. In other words, the litmus tests we take as input are basically of the shape $\forall$.

Definition 3.1. A litmus test is considered to be *well-defined* if its litmus condition is of shape $\forall$.

In order not to impose too much burden on the user side, we do not require the full state being specified in the litmus condition, as long as those major variables are covered. Our underlying machine learning based technique is able to tolerate some degree of missing information.

3.2 Memory Consistency Models

Existing memory consistency model specifications are formally defined either operationally (Flur et al. 2016; Sarkar et al. 2011; Sewell et al. 2010) or axiomatically (Alglave et al. 2014; Batty et al. 2016, 2011; Manson et al. 2005). Operational approaches are usually based on explicit notion of buffers, they define an abstrate state machine and specify what actions can happen at each step. On the contrary, the axiomatic approaches define the axioms that must hold throughout executions. Many existing axiomatic models define a model in two steps:

(1) Be a candidate execution. This is to ensure some fundamental safety properties on the execution so that the execution is meaningful.
(2) Be a valid candidate execution. The validity is the model-specific requirements on the execution. Acyclicity properties on complex derived relation compositions are usually used as the validity criterion.

The axioms in axiomatic memory model specification are properties over relations within each candidate execution. The types and semantics of of such relations are postponed to ξ4. Such specifications usually derive complex relation compositions from basic relations and check different axioms upon them. Most of such axioms

are talking about acyclicity properties.[1] Different memory consistency model formalizations may have different derived relations and different axioms.

In this work, we follow the the per-candidate-execution style axiomatic model. Our learned latent model is also based on a set of basic relations and properties upon them. The properties are only about being valid candidate executions. The fundamental safety properties to become candidate executions are not covered because they are shared across different models and does not need to be learned dynamically. After all, those are mainly sequential properties and it is the concurrent behaviors that different memory models vary.

By following the axiomatic style definition we also inherit the notorious Out-Of-Thin-Air (OOTA) issue, which has been shown as inevitable for axiomatic models (Batty et al. 2015). We do not consider OOTA treatments in this work.

4 APPROACH

In this section, we introduce our entire methodology of learning a latent memory model from litmus tests. It is very hard, if not impossible, to learn a latent memory model directly from litmus tests. We notice that one memory model is essentially featured by its many litmus tests; while one litmus test can be reasoned about through all its executions; and each execution can be validated according to its per-candidate-execution style axiomatic semantics. Hence, we convert the problem of "learning from litmus tests" into "learning from a collection of concrete executions".

4.1 Execution Graphs

Each concrete execution is represented as an *execution graph*, which contains a set of *memory events* as well as a set of partial order *relations* among those events.

Each memory event basically contains following attributes:

- ID, for identification;
- Thread ID;
- Action type such as Read, Write, Fence, or RMW (read-modify-writes);
- Address to Read or Write;
- Value read or written;
- Memory Order.[2]

A Relation $\langle rel, e_1, e_2 \rangle$ is simply some partial order of type rel over two memory events e_1 and e_2. There are basic relations with certain semantics as well as complex derived relations from basic ones. Take the happen-before relation *hb* for instance, *hb* for PowerPC is defined as "$preserved\text{-}po \mid fences \mid rf_e$" (Alglave et al. 2014). This specifies that

- those preserved program order relations
- those fences relations
- those read-from relations across different threads

are all treated as a happen-before relation. Then the model asserts that such *hb* relation set should always be acyclic.

In this work, we only take the basic relations plus one common derived relation as input because those derived relations and axioms require fundamental domain knowledge to define. Our learning framework is supposed to bypass those details as much as possible.

We assume that following relations are available in a concrete execution, and argue that these relations can be easily drawn using existing static analyses.

[1] There do exist non-acyclicity axioms in C11 about Non-atomic Read From Relations and Data Races (Batty et al. 2016).
[2] Only meaningful in C11.

po *program-order*, sometimes also called *sequenced-before*. *po* specifies the ordering of two events in the original sequential program. *po* relation information can be obtained from source code.

rf *read-from*. *rf* flows from a Write event W to a Read event R, specifying that the read value of R comes from W. In most created litmus tests, every write has a unique value (Wickerson et al. 2017) so that reads can easily match with writes according to the values they see. The *rf* relation information can thus be obtained.

co *coherence-order*, sometimes also called *modification-order* or *write-serialization*. *co* specifies a linearization order among all writes regarding the same address. The coherence axiom in some memory models needs this relation directly to state that different threads sees a coherent write order on the same address.

fr *from-read*, sometimes also called *read-before*. *fr* is defined as a Read event R being logically before another Write W event because it reads from another Write W' that comes before W in the coherence order. This is more formally defined as "rf^{-1}; *co*". *fr* is the only derived relation we take as input. It is widely used in real memory model specifications.

Fences and Barriers Various fences and barriers are used to restore consistency in architectures or languages, such as *mfence* in TSO, *lwsync* in Power, *relacq* in C11, etc.. They are written explicitly in programs and are given as events in the execution graph, but are then converted to relations in our learning framework. This conversion is to simplify their reasoning together with other relations.

Depencencies *Address*, *Data*, *Control*-dependencies may also appear in our executions since they are not derived from any basic relations. These dependencies provide more fine-grained control comparing to fences/barriers. These dependencies information can be obtained from basic static analyses.

4.2 Generate Labeled Executions

The next question is to generate those concrete executions such that they have expressed all weak memory behaviors, either allowed or forbidden, by the original litmus tests. We adapted a memory model simulator herd7 v7.44 (Alglave et al. 2014) to do the generation.

As a memory model simulator, herd7 is able to generate the complete set of executions against a particular memory model specification. Recall that our goal is to let produced executions contain all weak memory behaviors and use the original litmus test condition as an oracle to judge them. Hence, a weakest memory model is provided to the model simulator, so that it provides only sequential semantics correctness guarantee. The "uniproc.cat" shipped with herd7 is the weakest model we want.

The generated executions are then labeled positive or negative according to their original litmus condition. We require the litmus condition of input tests to cover all acceptable states, so that those satisfying (unsatisfying) the condition are roughly treated as allowed (forbidden) executions by the latent memory model.

The input litmus tests can thus be replaced by a large collection of labeled executions. Within the execution graph, we no longer need to worry about which program variable gets what wrong value so that litmus condition fails. Instead, we can focus on searching the corresponding relation pattern that forbids the entire execution.

4.3 Executions, Cycles, and Noise

Generated labeled executions will be fed to our learning framework to learn a latent memory model. We have implemented two versions of underlying learning framework to learn a latent memory model. The first one is based on Decision Tree (ξ5), and the other is using Conditional Random Field (ξ6). Although the underlying techniques may differ, the preprocessing and features used in two versions of learning framework are very similar.

In our learning framework, we want to learn the latent acyclicity properties used to define many existing memory consistency models. So simple cycles formed by relations are first extracted from executions using Tarjan's algorithm (Tarjan 1973). Every generated cycle path is simple in the sense that there is no repetition

on vertices or edges. Different relations connecting the same two events are merged as one edge in the cycle searching algorithm. They are restored after extracting the cycle.

If an execution contains no cycles, it is trivially deemed as allowed. This trivial check could be unsound when the actual model depends on non-acyclicity axioms. In this work, We only consider information within cycles so as to obtain pertinent features for a highly accurate classsifier. The learning, inference, and statistics discussed in our learning framework are only for those cycled executions.

Cycles extracted from positive executions are all treated as "allowed" otherwise one erroneous cycle should invalidate the entire execution. In the mean time, cycles extracted from negative executions need more care. There could be multiple cycles from one negative execution and it is unclear which ones are to blame. It is our goal to learn the latent memory model that judges which cycles exhibit the bad pattern.

To alleviate this issue, we apply a noise removal pre-processing procedure. The intuition is that if a cycle from some negative execution is observed in some positive execution, the negative cycle can be marked as noise and freed from later training. In the noise removal procedure, two cycles are considered to possess same pattern if there exists a one-to-one mapping in their events set. The one-to-one mapping in events requires a one-to-one mapping in threads and variables. The mapped events should have exactly the same actions and memory orders. It is fine for IDs and read/write values to vary, though. This alleviates the noise introduced by cycles in negative executions.

During training, the acyclicity axioms are reflected as global structural axioms on cycles. Our learning framework approximates them by local neighboring relation pairs. Further details of two learning frameworks will be discussed in $\xi 5$ and $\xi 6$, respectively.

4.4 Workflow

Therefore, our entire training workflow is summarized as:

(1) The input well-defined (Def 3.1) litmus tests are fed to our adapted herd7 to generate all possible executions on a weakest model which only guarantees correct sequential program semantics.
(2) All these weakest executions are classified according to their companion litmus conditions.
(3) Those labeled concrete executions containing cycles are fed into our underlying learning framework.

Our inference workflow is summarized as:

(1) Queries over a program property are reduced to queries over its executions.
(2) Assume the learned latent memory model is M. Note that cycles are subgraphs of executions so they have the same event-relation graph structure. M is able to infer on such graph structure.
(3) For every concrete execution G, extract all its cycles into a set C.
    - If C is empty, G is trivially deemed as allowed.
    - Otherwise, $M \vDash G$ only if $\forall$ cycle $c \in C$, $M \vDash c$.

4.5 Discussion

Currently, global structured properties are approximated by local relation pair features in both Decision Tree and CRF based classifiers. Some longer range relation compositions are thus not expressible, such as "propbase & WW" or those recursive "detour/cc" definition in PowerPC. Anyway, our goal is to approximate sophisticated specifications by simple learning approaches.

Besides, the current approach is inevitably limited by the expressiveness of memory event traces which contain only the simplest read write information.

5 DECISION TREE BASED LEARNING FRAMEWORK

In this section, we describe a decision tree based learning framework that takes a collection of labeled executions as input and learns a classifier.

We apply Decision Tree approach to learning the latent memory model from labeled executions/cycles because it does not suffer from the heterogeneity problem in learning specifications (Murali et al. 2017; Raychev et al. 2016). A specification, especially a complex one like that for memory consistency models, has to take various aspects into consideration. Doing one more feature splitting in Decision Tree will not affect globally since other paths of the tree remain unchanged.

In this work, the classical Decision Tree algorithm ID3 (Quinlan 1986) is implemented. The algorithm itself is easy, information gain is chosen to be the metric for selecting best features in tree spliting. The early stopping mechanism is adopted so that the tree stops growing when none of the available features is improving the information gain.

5.1 Decision Tree Features

It is known that Decision Trees have a high risk of overfitting. In our case, if the feature is defined as every structural detail of the cycle, it may easily fit the training set with a 100% correctness, but it doesn't help on the actual data. In this work, we approximate such global structural properties on cycles by local neighboring relation pair features. We select three categories of features:

(1) Does the cycle contain $rel_1 \cdot rel_2$ relation pair?
(2) Does the cycle contain $rel_1 \cdot rel_2$ relation pair with first relation having events pair features F?
(3) Does the cycle contain $rel_1 \cdot rel_2$ relation pair with second relation having events pair features F?

The first kind of features ask whether there exists two relations R_1 and R_2 such that R_1 is of type rel_1 and R_2 is of type rel_2, and that R_1 is immediately followed by a relation (R_1's tail event is the same as R_2's head event). Features (2) and (3) are based on feature (1) and are more restrictive than it.

In feature (2) and (3), the events pair features F for a relation $R = \langle rel, e_1, e_2 \rangle$ talks about attributes for (e_1, e_2), such as:

- Do e_1 and e_2 belong to same thread?
- What is the action pair for (e_1, e_2), is it Write-Write?
- What is the memory order pair for (e_1, e_2), is it Release-Acquire?[3]

These are all basic attributes of memory events. Intuitively, the ID and value attributes of memory events can be ignored. The address attribute in memory events is ignored as well, because they are mainly used in resolving relations like *rf*, *co*, or data dependencies. When such relations are available, the address information can be retired.

When two features results in equal metrics for splitting in decision tree, we prefer the simpler feature. This leans towards clearer hierarchy of features which may enable further specification reconstruction based on produced decision tree. Feature (1) is considered to be simpler than the other two because it has less constraints.

The events pair features must be prepared because they do appear in formalized memory consistency model specifications. Take the simplest TSO specification for exapmle, the "same-thread" feature is used to define rf_e relation which filters those *rf* relations with reads from external threads; and the action pair features are used to define the allowed reorderings in TSO.

The three categories of features are instantiated using all kinds of relation pairs recognized by the architecture/language and all types of event pair features. It's OK to have some features that do not make much sense,

[3]Only meaningful in C11.

like querying about a Fence relation with the same-thread events pair feature. Such bogus feature is unlikely to work better than other genuine features, thus unlikely to be selected into the final decision tree.

In fact, this is exactly our intuition of choosing Decision Tree approach as the underlying classifier. The basic relations for defining derived relations are finite. The composition operators being used to define a complex memory model specification are also finite. So when the number of maximum times of composition and the number of relation sequence is bounded, the entire state space for features is finite. According to experiments, the most basic pairwise relation features packed with event features already work well. Decision Tree is thus a good candidate for picking the most pertinent features among that state space.

The three categories of features we defined are able to approximate most of the "features" in many well-defined specifications. Take the framework in (Alglave et al. 2014) for example, most of its connectives used in formalization can be approximated by our simple features:

- $\&$ conjunction, can be approximated by feature (2) or (3).
- $|$ disjunction, can be approximated by multiple of our features.
- $\cdot$ sequential composition, can be approximated by multiple of our features.
- $\setminus$ exclusion, can be approximated by the negation of our features.
- $^+$ transitive closure, usually used together with empty or acyclic keywords to assert the acyclicity of a cycle. This is approximated when we query "contains feature inside the cycle?".
- $^{-1}$ inverse, this is not represented by our features since our relations are directed and the direction remains untouched. Nonetheless, we argue that the inverse operator is rarely used, we have only observed one such occurrence in C11 model besides relation *fr*. Relation *fr* has been explicitly taken into consideration.

With the features defined above, a cycle satisfies a feature if certain types of relations pair with certain events pair feature is observed in the cycle. Recall that cycles from negative executions are problematic because there is no accurate labels for them unlike positive cycles. To deal with this, we unfold all cycles in positive executions but maintain the negative executions as they are. The feature query procedure for negative executions is also modified. A negative execution satisfies a feature as long as there exists one cycle from within that satisfies the feature. This will inevitably make features less precise in negative data. But comparing to unfolding negative cycles and assign negative labels to them, we choose to live with imprecise attributes rather than inaccurate labels. When doing inference, all cycles are still unfolded and inferred individually.

6 CONDITIONAL RANDOM FIELD BASED LEARNING FRAMEWORK

In this section, we describe another Conditional Random Field (CRF) (Sutton and McCallum 2012) based learning framework. We noticed that the latent acyclicity properties are global structural properties on executions/cycles. This lends itself to a Structured Prediction problem in which CRF has been very active.

6.1 CRF overview

CRF is an undirected discriminative graphical model that has been successfully applied in many areas. Here we briefly introduce the formulation and idea behind CRF. Readers are referred to (Koller and Friedman 2009; Sutton and McCallum 2012) for further details.

CRF is usually constructed upon factor graphs which can also subsume other graphical models such as Baysian Network or Markov Random Field. Factor graphs contain random variable nodes (usually drawn as ○) and factor nodes (usually drawn as □), in which factor nodes are used to connect different variable nodes, as shown in Fig. 1.

CRF is essentially a Markov Network with a directed dependency on some subset of random variables. It is defined with respect to *factors*. The probability distribution of CRF is generally defined as:

$$P(Y|X) = \frac{1}{Z(X)} \prod_{i=1}^{m} \phi_i(D_i)$$

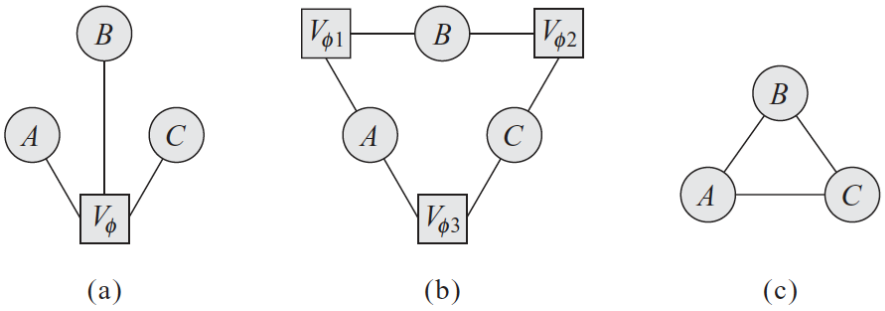

Figure 4.8 Different factor graphs for the same Markov network: (a) One factor graph over A, B, C, with a single factor over all three variables. (b) An alternative factor graph, with three pairwise factors. (c) The induced Markov network for both is a clique over A, B, C.

Fig. 1. Different factor graphs example, clipped from (Koller and Friedman 2009).

where $Z(X)$ is the partition function:

$$Z(X) = \sum_{Y} \prod_{i=1}^{m} \phi_i(D_i)$$

In this formulation, $X \cup Y$ is the entire input random variables while X is the observed set and Y is the set to predict. Each D_i is a *factor*, i.e., a complete subgraphs of original graph. There are m such factors enumerated. Different problem instantiations may divide different factors for their own purpose. It is required that the factor cannot be those variables only in X otherwise Y is not affected. In this way, CRF models those nodes that are relevant for predicting Y instead of every joint distribution of edges in X. Modeling entire features could be very difficult, CRF provides a way to decompose large graph into small factors.

$\phi_i()$ computes the "affinity" inside D_i, i.e., how compatible the nodes in D_i are. The actual computation of $\phi_i()$ also varies in different applications. $Z(X)$ sums on all cases of Y. This is used to normalize computed probabilities.

In reality, another more compact formulation is usually used to simplify computations under the hood. The Π operator is replaced by exp() and $\sum$ operator:

$$P(Y|X) = \frac{1}{Z(X)} \exp(\Phi_i(X, Y))$$

where

$$\Phi_i(X, Y) = \sum_{i=1}^{m} \phi_i(D_i)$$

$$Z(X) = \sum_{Y} \Phi_i(X, Y)$$

Exact inference is very hard for general CRF formulations, because the number of posterior distribution components is exponential to the size of training data. In this work we use one approximate inference algorithm – sum-product Loopy Belief Propagation (LBP) (Koller and Friedman 2009). LBP is simply Belief Propagation algorithm being applied in a loopy setting. It is able to approximate the probability of individual random variables, via the computation of *belief* in each node.

Basically, in each iteration of LBP, nodes propagate beliefs to neighbors as well as receiving updates from neighbors and adjust own beliefs accordingly. This process keeps on going until it eventually stabilizes or the specified iterations bound is exceeded. Each belief message $m_{i \to j}$ contains a size-k vector where k is the possible categories node j belongs to. The message contains the beliefs from node i about node j being in each of these categories. The belief of a variable node is summarized from its neighboring factor nodes. Similarly, the belief of a factor node is summarized from its neighboring variable nodes. The affinity score $\phi_i(D_i)$ is used in computing the belief message from factor subgraph D_i's corresponding factor nodes.

During training, LBP is also used as the training algorithm may need to call inference algorithm multiple times. Training is formulated as a Maximum Likelihood Estimation (MLE) problem and solved by Adaptive Gradient algorithm (Duchi et al. 2011) with parameter averaging.

In our implementation, the detailed computations in CRF are delegated to FACTORIE (McCallum et al. 2009), we only define how to construct factors and how to compute the affinity scores among them.

6.2 Problem Formulation in CRF Graph Structure

The graph structure in CRF is indeed decomposing the features we want to learn. Realizing that relations are more important than memory events in executions/cycles, we make relations as variable nodes and memory events as factor nodes. Memory event factor nodes serve as connectors between different relation variable nodes, no other information is stored in them.

During inference, a new execution/cycle is provided to query about its correctness regarding the learned latent memory model. Some relations like *po* are trusted while some relations like *rf* are indeed what we want to reason about. We call the former category of relations *trusted* and the latter *unfixed*. Such unfixed relations are the Y set in CRF formulation. Then correctness query over entire execution/cycle is reduced to the query whether each such unfixed relation can appear given all other relations observed in the graph. We categorize *rf*, *fr* as unfixed and all other relations are trusted.[4] *fr* is also unfixed because it is constructed depending on *rf* and it is possible that some cycle contains *fr* relation but not *rf*.

There could be multiple relations connecting the same two events, representing all of them individually would make the generated CRF graph like a large spider web. Such complex graph structure would make it very difficult to do approximate inference in CRF. Hence, the relations touching the same two events are merged into a bitset and encoded as a binary number. Thereafter, the entire cycle path can be encoded as a single cycle without any forks. The marginal inference over this bitset encoding is also the reason we choose sum-product version LBP instead of max-product LBP.

However, this raises another problem when doing inference. When all relations are merged into one bitset, the encoded number would be in the range of $[0, 2^k)$ when there are k different relations to consider. In complex models like PowerPC, the value of k could be as large as 11. In this case, inferring the probability of unfixed relations would be greatly affected by other trusted relations encoded together.

Therefore, we separate the trusted and unfixed relations set between every two memory event factor nodes. Hence, every memory event factor node may have a collection of predecessor relation variable nodes and a collection of successor relation variable nodes. Every distinct pair of predecessor and successor is selected as a factor subgraph D_i as long as they are not both trusted relation sets. There is a tradeoff between the complexity of constructed CRF graphs and the relevance of inference on individual relation variable nodes. The current configuration turns the CRF graph into a twisted-wire-like shape. The empty relation set will be pruned to make the constructed CRF graph as succinct as possible.

[4] This brings one problem in ARM model, where 2+2W test series may have cycles without either *rf* or *fr*. Thus they are not supported by this categorization.

There is another discrepancy that relations are directed while CRF is inherently undirected. When selecting factor subgraphs, we strictly follow the relation pair flow order so that affinity score computation in each factor subgraph is ordererd.

In addition to relation variable nodes, the attributes of each relation should also be taken into consideration. The actual features for events pair features for each relation are exactly those defined in ξ5.1 for Decision Tree. The difference is that such events pair features are represented as another type of variable nodes in CRF graph. Those variable nodes contain binary feature vectors of such feature information. Each relation variable node will be connected to one such feature variable node. Every such relation variable node with its feature variable node pair will form a factor subgraph as well.

So far we have defined two types of factor subgraphs:

(1) Between two relation variable nodes;
(2) Between one relation variable node and its corresponding feature variable node.

We only use these two types of pairwise factors in the framework. For each type of factor subgraph, a 2-dimension matrix M_0 is prepared as weights matrix such that M_0's size matches the outer product of two variable nodes' sizes. To compute the affinity score $\phi_i(D_i)$, the concrete categories of two variable nodes are first collected as the same size outer product matrix M. Then the dot product between M and weights matrix M_0 is used as the affinity score. The weights matrix will be optimized during training automatically.

Intuitively, this weight matrix contains the multiplicative factor for every pair of concrete instance. For example, when a rf_e relation is observed in a forbidden cycle, the optimization process during training might descrease the corresponding weight of "external" feature related to relation *rf*, thereby reducing the likelihood of a cycle with rf_e being accepted.

6.3 Noise Negative Data

Different from Decision Tree based classifier, cycles in negative executions must be unfolded before training in CRF. This is because CRF naturally treats all training data as positive so the bad parts in negative cycle must be explicitly specified in training. Such error marking is not possible if the negative cycles remained together like that in Decision Tree. In the mean time, all cycles in positive data are still considered as allowed individually.

Knowing that litmus tests are usually very small, we assume that by removing one concrete *rf* relation it should break the erroneous cycle and make the entire execution correct again. Hence, for each such *rf* relations in the original negative cycle, we flip *rf*, marking it as no-*rf* in the same location and produce a new concrete cycle. If there is no *rf* in the cycle, *fr* relations are flipped accordingly. With such flipping, we expect the classifier is trained to learn that no-*rf* is more probable than *rf* in certain forbidden scenarios.

In addition to the noises over which negative cycle is the bad pattern, this flipping procedure inevitably introduces more noises over which particular *rf* relation is indeed forbidden. The relatively large amount of generated cycles may also impact the positive/negative data distribution.

To alleviate this *data imbalance* and *noisy data* issue, we first remove any redundant training data, because many generated training data through the above process are actually identical.

Then we apply Bagging (Breiman 1996) to resample different replicas of the pruned training data. The intuition is to try different variants of data so that the noise only appear in some replicas but not all of them. The inference result is the majority vote of classifiers learned for each replica. During replica preparation, positive data and negative data are resampled individually so that the data distribution is maintained. If the distribution is too imbalanced, under-sampling is applied in bagging replica sampling so that the majority class of data only draws the same amount of data as that in minority class of data.

6.4 Inference

When doing inference on a concrete execution, we check if all its cycles are accepted by the learned model. For each cycle, we check if all its unfixed relations are predicted with higher probability of existence. Recall that relations touching the same two events are merged into a bitset to reduce the graph complexity. We compute the prediction of an unfixed relation R residing in variable node N by following steps:

(1) Given the observation of other relations in N, we first compute the probability of N having R as p_t;
(2) Then we compute the probability of N not having R as p_f;
(3) The prediction probability is computed as $p_t/(p_t + p_f)$.

This normalization step is necessary because there could be other relations encoded in N as well.

Different training data may impose different bounds for a computed probability to be accepted. This bound is selected using validation set by comparing every integer bound from 1% to 99%. The selection strategy can be optimized in order to maximize certain metrics. The default option is to $\max\{\min\{F1, F1_N\}\}$ where $F1_N$ is the F1 score computed on negative data. This is to search for the classifier that performs good enough on both positive and negative data.

6.5 Workflow

To conclude, the workflow for our CRF-based learning framework is:

(1) During Training:
  (a) Use only those cycled executions in training.
  (b) Cycles from positive executions are used directly. Cycles from negative executions undergo a flipping procedure to mark the potential forbidden relations.
  (c) Remove noise and redundancy detected in training data.
  (d) Feed the remaining training data to CRF classifier for training using the graph structure, features, algorithms discussed above.
(2) During Inference:
  (a) Extract cycles from execution, if there is no cycle, it's trivially deemed allowed.
  (b) Convert every extracted cycle path to CRF graph.
  (c) Infer on the cycle path CRF graph over all its unfixed relations.
  (d) Cycle is accepted only if all such unfixed relations are predicted with a high enough probability.
  (e) Execution is accepted only if all its cycles are accepted.

6.6 Discussion

FACTORIE is chosen because it is still being maintained and claims to support various CRF graph structures. Although at the end we only use pairwise factors. The current problem formulation is chosen because that fits FACTORIE framework and LBP algorithm quite well.

Besides noise removal by exact matching, I had also tried other approaches such as "voter filtering", which makes multiple replicas like those in cross validation or bagging and mark a data point as noise when all trained replicas say so. But this doesn't improve much.

6.6.1 Why CRF?. In fact, we first implemented the CRF based classifier. It is only after revisiting the finiteness property on possible features over input executions that we start to try Decision Tree based approach. The skip-chain CRF graph structure is very similar to the cycles extracted from executions, which adds some confidence to the CRF technique being successful.

Another original reason for choosing CRF but not directed graphical models such as Bayesian Network is that they do not accept cycles. Although now I realize that it is possible to break cycles into straight sequences.

6.6.2 Why no CRF?. From the lengthy discussion above you may have realized that formulating the problem uisng CRF requires non-trivial efforts. However, it does not lead to better results in our experiments. Actually it performs worse in most cases comparing to the much easier Decision Tree based classifier.

Originally, I was hoping that structured prediction techniques being applied to program analysis products will be helpful. This is later refuted by experiments. The formulation in CRF may still be improved. Although they may be relevant, the events pair features of a trusted node will not affect the reasoning of unfixed nodes at all because they are not considered by CRF implementation in FACTORIE.

Nonetheless, the biggest reason is hypothesized to be the heterogeneity problem. Now that the same weights matrix defined in CRF are used in every occurrence. The weights update are global unlike that in Decision Tree. When the specification to learn is complex enough, multiple aspects must be taken into consideration. Then learning one aspect may distract the learning in other aspects, especially when we do not have abundant data.

We have already extracted cycles from executions but not use the executions directly. But maybe that is still not enough, since many nodes in current CRF that are irrelevant for memory model acceptance are still used in training. Although it is our goal to learn which nodes are bad and which are irrelevant. This may explain why CRF is a good candidate for predicting JavaScript properties in (Raychev et al. 2015) – the nodes there are real features.

7 EXPERIMENTS

To demonstrate the feasibility of our approach, experiments have been conducted on TSO, PowerPC, ARM, and part of C11. We have collected various litmus tests for each architecture/language and instrumented their litmus conditions with all accepted final states using existing formalized model. The labeled executions are generated through our adapted herd7 using the "uniproc.cat" weakest model specification. Among all generated labeled executions, we did 10-fold cross validation with 8 pieces of training data, 1 piece of validation data, and 1 piece of test data. Our goal is to demonstrate that the learned latent memory model is able to predict on test data highly accurately. The statistics details regarding all cycled execution are reported.

7.1 Dataset Preparation

Our experiments cover TSO, PowerPC, ARM, and part of C11. Table 1 shows the formal model we use to generate litmus conditions for each architecture/language that enumerates all allowed states. All formal models are shipped with herd7 tool itself. The model used in C11 experiments comes from (Batty et al. 2016). The relations observed and handled for each model are also listed in the table.

Experiment	Formal model	Observed Relations
TSO	x86tso.cat	*po*, *co*, *rf*, *fr*, *mfence*
Power	ppc.cat	*po*, *co*, *rf*, *fr*, data/addr/ctrl dependency, *sync*, *lwsync*, *isync*, *ctrlisync* barriers
ARM	arm.cat	*po*, *co*, *rf*, *fr*, data/addr/ctrl dependency, *dmb*, *dsb*, *isb*, *ctrlisb* barriers
C11	c11_partialSC.cat	*po*, *co*, *rf*, *fr*, data/ctrl dependency, *sc*, *acq*, *rel*, *acqrel*

Table 1. Architecture/Language Configurations

The datasets for each architecture/language are shown in Table 2. The 3rd column shows the entire amount of collected tests. The 4th column shows the number of executions containing cycles since they are the sources of training data. The last column shows the number of unfolded and cleansed cycles from those executions. Noise removal and redundancy removal have been applied in cleansing. Note that some datasets are very imbalanced.

Dataset	Sources	#Tests	+/− cycled executions	+/− cycles after cleansing
TSO ESOP'13	(Alglave et al. 2013)	114	82 / 114 41.84% / 58.16%	73 / 51 58.87% / 41.13%
Power ESOP'13	(Alglave et al. 2013)	443	608 / 398 60.44% / 39.56%	434 / 205 67.92% / 32.08%
Power PLDI'11	(Sarkar et al. 2011)	764	692 / 291 70.40% / 29.60%	562 / 331 62.93% / 37.07%
Power CAV'10	(Alglave et al. 2010)	770	724 / 3396 17.57% / 82.43%	334 / 1264 20.90% / 79.10%
Power merged	all Power sources above	1775	1840 / 3701 33.21% / 66.79%	1105 / 1540 41.78% / 58.22%
ARM PLDI'11	(Sarkar et al. 2011)	350	265 / 124 68.12% / 31.88%	238 / 183 56.53% / 43.47%
C11 merged	(Batty et al. 2016) (Vafeiadis et al. 2015) herd7 old testsuite	489	202 / 180 52.88% / 47.12%	128 / 267 32.41% / 67.59%

Table 2. Dataset Distribution

For C11 experiments, we collected tests from multiple sources because each source itself is not enough to do meaningful experiments. The tests from (Vafeiadis et al. 2015) are collected from a github repository that generates tests and many variants from that paper. The tests for C11 are instrumented to use every location in program as litmus condition because the conditions original litmus tests are poorly specified. Litmus tests from other datasets are in a much better shape and do not need such pre-processing. Regarding the non-acyclicity properties in C11, about 20 executions are affected by them.

7.2 Statistics for Decision Tree based classifier

Table 3 shows the detailed metrics for our Decision Tree based classifier in every dataset mentioned above. It removes negative noises by exact match in positive data but no redundancy removal. Now that cycles in negative executions are not unfolded, redundancy removal in negative data will lose important information.

In the table, "Accuracy" is the correctness on all data; "Recall" is the correctness on positive data while "Specificity" is the correcteness on negative data. "F1" is the harmonic mean of precision and recall on positive data while "$F1_N$" is the F1 score computation on negative data.

The numbers for C11 could be improved even more when we also add those in-cycle READs' corresponding WRITES into cycle. Those WRITEs may not be in the cycle but are actually considered used in existing C11 formal models. The statistics in this updated setting is shown in Table 4.

To summarize, preliminary experiments have shown that the accuracy and F1 score of our learned model can be as high as 95%+.

Dataset	Accuracy	Recall	Specificity	F1	$F1_N$
TSO ESOP'13	100%	100%	100%	100%	100%
Power ESOP'13	95.58%	96.468%	94.299%	96.297%	94.479%
Power PLDI'11	77.982%	87.984%	53.755%	84.888%	58.508%
Power CAV'10	98.904%	95.667%	99.591%	96.849%	99.336%
Power merged	95.53%	91.683%	97.445%	93.176%	96.672%
ARM PLDI'11	75.022%	88.517%	48.1%	83.095%	53.436%
C11 merged	90.377%	95.443%	84.292%	91.412%	88.632%

Table 3. Statistics for Decision Tree based classifier

Dataset	Accuracy	Recall	Specificity	F1	$F1_N$
TSO ESOP'13	100%	100%	100%	100%	100%
Power ESOP'13	95.586%	96.199%	94.783%	96.307%	94.48%
Power PLDI'11	77.371%	87.596%	53.057%	84.482%	57.626%
Power CAV'10	97.646%	90.513%	99.177%	93.086%	98.58%
Power merged	94.801%	90.378%	97.018%	92.038%	96.136%
ARM PLDI'11	75.558%	90.536%	45.075%	83.395%	51.832%
C11 merged	94.009%	96.769%	90.741%	94.536%	93.105%

Table 4. Statistics for Decision Tree based classifier when corresponding WRITEs are also added to cycle

7.3 Statistics for CRF based classifier

Table 5 shows the detailed metrics for our CRF based classifier in every dataset mentioned above. Using CRF as the base classifier, it removes negative noises by exact match in positive data and removes redundant data in all data. Bagging is enabled with 50 iterations, and under-sampling is used when the data distribution is too imbalanced. The MaxMinF1F1$_N$ strategy is chosen for model selection against validation set data.

Dataset	Best Bound	Accuracy	Recall	Specificity	F1	$F1_N$
TSO ESOP'13	88%	86729%	84.694%	88.64%	85.32%	87.431%
Power ESOP'13	82%	88.203%	84.815%	93.199%	89.547%	86.29%
Power PLDI'11	69%	80.426%	76.485%	90.216%	84.571%	72.887%
Power CAV'10	76%	85.648%	56.026%	91.911%	57.912%	91.32%
Power merged	67%	84.757%	73.732%	90.261%	76.257%	88.766%
ARM PLDI'11	79%	78.419%	73.516%	89.77%	82.02%	71.746%
C11 merged	55%	72.823%	67.887%	77.802%	71.87%	72.991%

Table 5. Statistics for CRF based classifier

The 2nd column "Bound" is the probability bound such that only prediction higher than this bound is accepted by the model. This bound is selected according to validation set automatically.

Different from Decision Tree, the "add corresponding WRITEs to cycle" update does not help CRF-based classifier because it actually increases the CRF graph complexity and affects training and inference.

7.4 Discussion

7.4.1 Regarding Datasets. For TSO, the "xchg" tests from white paper are ignored because herd7 does not seem be support that. Anyway, I was eventually using the ESOP'13 dataset.

For PSO, I also thought about PSO experiments, but there is no abundant PSO dataset.

Actually, I thought about one alternative of feasibility demonstration method:

- Instead of using the executions from "uniproc.cat" for training data, use the null model which has no constraints at all.
- Instead of using the executions from "uniproc.cat" for test data, use those allowed by the real formal model as positive data, and those rejected by the null model as negative data.

Apparently, the labels in test data will be guaranteed to be accurate. Cons of this new demonstration methodology could be:

(1) The training is much more noisy where it also needs to handle sequential error as well. Some large litmus test may even timeout during executions generation.
(2) Using positive and negative executions from different sources makes it hard to do cross validation. As a result, some test data executions may appear in training data.

7.4.2 Regarding Results. For Decision Tree based classifier, the numbers on Power PLDI'11 and ARM PLDI'11 dataset are much worse then other datasets. Both datasets come from (Sarkar et al. 2011) and are generated in a similar way, we do not know why those datasets are less compliant with our Decision Tree approach. These two datasets are also the only places the CRF version classifier outperforms Decision Tree version.

8 RELATED WORK

8.1 Synthesizing memory models from litmus tests

There are several works that tackles the similar problem in different approaches (Alglave and Maranget 2011; Bornholt and Torlak 2017; Wickerson et al. 2017).

(Alglave and Maranget 2011) is one tool whose exploration mode claims to explore a real machine automatically and outputs the safe/forbidden relation cycle sets. This short tool report has been discarded, its code has been removed from their latest version framework.[5]

(Bornholt and Torlak 2017) is also addressing the problem of generating memory model specifications out of litmus tests. Different from us, they require some template to be provided as sketch for synthesis. Then everything is encoded into formulas in their bounded relational logic and delegated to the Rosette framework. (Wickerson et al. 2017) is another new paper that uses executions to represent a litmus program, and encode everything into some SAT formulas. These formulas are fed to SAT solvers to answer four types of different questions about comparing memory models. Comparing to these SAT/SMT-based approach, our approach requires much less domain knowledge to be provided and produces a discriminative model, which is more programmer friendly and can provide strong extensibility to other domains such as weak consistency in distributed systems. Moreover, traditional synthesis approaches generally are not good at noisy inputs.

8.2 Machine Learning techniques in PL community

The problem of learning a latent memory model from litmus tests can also be viewed as a synthesis problem whose output is a DSL problem that describes the underlying memory consistency model. Various techniques have been proposed for program synthesis (Alur et al. 2013; Gulwani 2011; Gulwani et al. 2015).

Recently, there has been a large amount of work to use machine learning techniques to help synthesis. Neural Program Synthesis works (Parisotto et al. 2016; Riedel et al. 2016) represent programs in a differentiable manner

[5]Relevant codes are last seen in version 5.01, last modified time 18-Sep-2015 14:55.

and uses gradient based search methods to do synthesis. Neural Program Induction works (Neelakantan et al. 2015; Reed and de Freitas 2015) learns a latent model and generates output directly without human interpretable model like that in synthesis problems. Our CRF based classifier is similar to neural program induction works but uses a very different technique. Our Decision Tree based classifier is somewhat between neural program synthesis and neural program induction. The produced decision tree is neither a complete blackbox nor a well-baked specification.

As shown in (Gaunt et al. 2016), it may not help much to apply gradient based learning techniques to a finite problem such as synthesis given template sketches. Learning can also be used to tune the best configuration of traditional methods for certain input data (Balog et al. 2016; Chae et al. 2016).

In addition to general synthesis problem, learning techniques have been actively incorporated into many other topics such as type or shape specification learning (Zhu et al. 2015, 2016), invariant inference (Garg et al. 2016; Sharma et al. 2013), super-optimization (Schkufza et al. 2013), probabilistic code model (Bielik et al. 2016; Raychev et al. 2016, 2015, 2014). In particular, (Raychev et al. 2015) is also using CRF to predict the properties of programs after obfuscation.

8.3 Learning from traces

Our approach is to learn from concrete execution graphs. Such graph-based approaches have been proved to be helpful in program analysis as well, such as anomaly detection (Akoglu et al. 2015) and fault localization (Liu et al. 2005). There are plenty of works learning properties from concrete execution traces (Gupta et al. 2015; Mühlberg et al. 2015; Weeratunge et al. 2011). Specification mining works (Talupur et al. 2015; Udupa et al. 2013) are also similar in this sense.

9 CONCLUSION

In this paper, we have presented a novel approach to learn latent memory models from a collection of litmus tests based on a provided weakes model. Our approach is based on the insights to reduce litmus tests to concrete executions that expose all allowed and forbidden weak memory behaviors by the latent model, and then approximate global structural properties by local neighboring features. Experiments have demonstrated the feasibility of our approach. Our approach can extend to various models due to its few requirements on necessary domain knowledges, as long as the models are defined by acyclicity axioms. As far as we know, this is the first approach that can apply to such a wide range of models automaticlly.

For future work, we plan to apply the approach to more user-specified/bespoke applications and provide more accommodations while still ensuring the properties desired by user.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

2013. *12 Million Loss due to weak memory behavior.* http://stackoverflow.com/questions/16159203/.

Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *Data Min. Knowl. Discov.* 29, 3 (2015), 626–688. DOI: http://dx.doi.org/10.1007/s10618-014-0365-y

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015.* 577–591. DOI: http://dx.doi.org/10.1145/2694344.2694391

Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software Verification for Weak Memory via Program Transformation. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 512–532. DOI: http://dx.doi.org/10.1007/978-3-642-37036-6_28

Jade Alglave and Luc Maranget. 2011. Automatic Weak Memory Model Exploration. http://diy.inria.fr/dont/final.pdf. (2011).

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. 258–272. DOI: http://dx.doi.org/10.1007/978-3-642-14295-6_25

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 7. DOI: http://dx.doi.org/10.1145/2594291.2594347

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 1–8. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679385

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *CoRR* abs/1611.01989 (2016). http://arxiv.org/abs/1611.01989

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 634–648. DOI: http://dx.doi.org/10.1145/2837614.2837637

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 283–307. DOI: http://dx.doi.org/10.1007/978-3-662-46669-8_12

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 55–66. DOI: http://dx.doi.org/10.1145/1926385.1926394

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2933–2942. http://jmlr.org/proceedings/papers/v48/bielik16.html

James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '17, Barcelona, Spain - June 18 - 23, 2017*. 13.

Leo Breiman. 1996. Bagging Predictors. *Machine Learning* 24, 2 (1996), 123–140. DOI: http://dx.doi.org/10.1007/BF00058655

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2016. Automatically generating features for learning program analysis heuristics. *CoRR* abs/1612.09394 (2016). http://arxiv.org/abs/1612.09394

David Dice. 2009. A race in LockSupport park() arising from weak memory models. https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park. (2009).

John C. Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research* 12 (2011), 2121–2159. http://dl.acm.org/citation.cfm?id=2021068

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 608–621. DOI: http://dx.doi.org/10.1145/2837614.2837615

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 499–512. DOI: http://dx.doi.org/10.1145/2837614.2837664

Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. TerpreT: A Probabilistic Programming Language for Program Induction. *CoRR* abs/1608.04428 (2016). http://arxiv.org/abs/1608.04428

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 317–330. DOI: http://dx.doi.org/10.1145/1926385.1926423

Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin G. Zorn. 2015. Inductive programming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99. DOI: http://dx.doi.org/10.1145/2736282

Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. 2015. Succinct Representation of Concurrent Trace Sets. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 433–444. DOI: http://dx.doi.org/10.1145/2676726.2677008

Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11886

Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. 2005. Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005*. 286–297. DOI: http://dx.doi.org/10.1137/1.9781611972757.26

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 378–391. DOI: http://dx.doi.org/10.1145/1040305.1040336

Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. A tutorial introduction to the ARM and POWER relaxed memory models. *Draft available from http://www. cl. cam. ac. uk/˜ pes20/ppc-supplemental/test7. pdf* (2012).

Andrew McCallum, Karl Schultz, and Sameer Singh. 2009. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*. 1249–1257. http://papers.nips.cc/paper/3654-factorie-probabilistic-programming-via-imperatively-defined-factor-graphs

Jan Tobias Mühlberg, David H. White, Mike Dodds, Gerald Lüttgen, and Frank Piessens. 2015. Learning Assertions to Verify Linked-List Programs. In *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*. 37–52. DOI: http://dx.doi.org/10.1007/978-3-319-22969-0_3

Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Finding Likely Errors with Bayesian Specifications. *CoRR* abs/1703.01370 (2017). http://arxiv.org/abs/1703.01370

Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. 2015. Neural Programmer: Inducing Latent Programs with Gradient Descent. *CoRR* abs/1511.04834 (2015). http://arxiv.org/abs/1511.04834

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-Symbolic Program Synthesis. *CoRR* abs/1611.01855 (2016). http://arxiv.org/abs/1611.01855

J. Ross Quinlan. 1986. Induction of Decision Trees. *Machine Learning* 1, 1 (1986), 81–106. DOI: http://dx.doi.org/10.1023/A:1022643204877

Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 731–747. DOI: http://dx.doi.org/10.1145/2983990.2984041

Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 111–124. DOI: http://dx.doi.org/10.1145/2676726.2677009

Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 44. DOI: http://dx.doi.org/10.1145/2594291.2594321

Scott E. Reed and Nando de Freitas. 2015. Neural Programmer-Interpreters. *CoRR* abs/1511.06279 (2015). http://arxiv.org/abs/1511.06279

Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. 2016. Programming with a Differentiable Forth Interpreter. *CoRR* abs/1605.06640 (2016). http://arxiv.org/abs/1605.06640

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 175–186. DOI: http://dx.doi.org/10.1145/1993498.1993520

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. 305–316. DOI: http://dx.doi.org/10.1145/2451116.2451150

Jaroslav Sevcík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. 27–51. DOI: http://dx.doi.org/10.1007/978-3-540-70592-5_3

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. DOI: http://dx.doi.org/10.1145/1785414.1785443

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013. Verification as Learning Geometric Concepts. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 388–411. DOI: http://dx.doi.org/10.1007/978-3-642-38856-9_21

Charles A. Sutton and Andrew McCallum. 2012. An Introduction to Conditional Random Fields. *Foundations and Trends in Machine Learning* 4, 4 (2012), 267–373. DOI: http://dx.doi.org/10.1561/2200000013

Murali Talupur, Sandip Ray, and John Erickson. 2015. Transaction Flows and Executable Models: Formalization and Analysis of Message passing Protocols. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*. 168–175.

Robert Endre Tarjan. 1973. Enumeration of the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 2, 3 (1973), 211–216. DOI: http://dx.doi.org/10.1137/0202017

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 287–296. DOI: http://dx.doi.org/10.1145/2462156.2462174

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 209–220. DOI: http://dx.doi.org/10.1145/2676726.2676995

Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2011. Accentuating the positive: atomicity inference and enforcement using correct executions. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 19–34. DOI: http://dx.doi.org/10.1145/2048066.2048071

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 190–204. http://dl.acm.org/citation.cfm?id=3009838

He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Learning refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 400–411. DOI: http://dx.doi.org/10.1145/2784731.2784766

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically learning shape specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 491–507. DOI: http://dx.doi.org/10.1145/2908080.2908125