

# Symbolic Extrapolation for Automata

June 3, 2008

## Slide 1-5: Logo Slides

Good afternoon. My name is Vijay. I am a PhD student and I shall leave the work of guessing my affiliation as an audience exercise. I'll slowly reveal more information about my affiliation here and you can put your hands up if you think you have guessed it.

The main idea of my talk is similar. We have a computation which we encounter during program analysis or verification. We would like, with a finite amount of data to be able to guess the final result and jump directly to it.

## Slide 6: Title Slide

This is joint work with Daniel Kröning, David Basin and Felix Klaedtke in ETH Zurich, and Daniel Kroening in Oxford. Even before giving you a talk outline, I will just review three basic concepts in program analysis to establish the vocabulary we will use in this talk.

## Slide 7: Program analysis 101: Abstract

From fundamental results of Gödel and Turing, made explicit in Rice's 1953 theorem we know that any non-trivial analysis of a Turing-complete programming language is undecidable. Even if we restrict the programming language, most program analyses are computationally hard.

The idea behind abstract program analysis is that if you ignore some details about the program, one can at least obtain approximate answers. Cousot and Cousot in their 1977 paper formalised the notion of abstract program analysis and conditions under which an approximate solution give us information about the program.

The analogy to image processing, which I have on my slide is not as far fetched as it may seem. The question of whether the Mona Lisa smiles has intrigued art enthusiasts for a few centuries. Supposedly, if you don't look at her mouth directly, you will see the smile. From work in visual cognition, we know that our peripheral vision is more sensitive to low spatial frequencies. Morphological filtering, which is part of what Margaret Livingstone, a neurobiologist at Havard, used to investigate this bit of folklore can and has been formalised completely within abstract interpretation.

## Slide 8: Program analysis 101: Infinite

If the abstract analysis is to be feasible, does the abstraction have to be finite? This is what a lot of people believed in the 70s, and in fact, some do so even today in the more compiler oriented areas of program analysis.

The problem is that no finite abstract domain is good enough for all programs. Infinite abstract domains are in general more expressive. But everything has a price, and that includes infinity.

Most program analysis methods can be viewed as iteratively computing a fixed point of a monotone function. One obvious problem with infinite domains is that this iteration process may not terminate. The other problem is that the function may not even have a fixed point.

For example, it is quite common to capture the relationship between numeric variables using polyhedra. The polyhedra may look like this. I could have a sequence of polyhedra whose limit is actually not a polyhedra. This may occur if they limit of the polyhedra is a sphere.

How do we cope with infinite domains? Cousot and Cousot suggested using a widening operator. What you do is guess the direction in which the analysis is progressing and then generalise. But they

didn't tell us how. That is what I will focus on in this talk. But first, let me introduce the third basic concept of program analysis.

### **Slide 9: Program analysis 101: Symbolic**

The third aspect, particularly from a practical perspective is how we represent abstract facts the analysis computes. A list is inefficient, and if you have an infinite set of elements, it's not even feasible.

A symbolic representation is a data structure to represent sets of facts. For example, a set of machine integers can be viewed as a set of bit-vectors. You can then think of these as defining a Boolean function. The function is true when its variables have exactly the values of one of the bit-vectors. That's the idea behind symbolic representations like BDDs or propositional logic formulae.

Now, I want to emphasise that an abstract domain and a symbolic representation are different. Propositional logic formulae could be used as an abstraction, but I can represent them using BDDs or CNF or DNF. Regular languages are another abstraction and I can represent them using finite automata, or regular expressions, or a system of equations.

The crucial point is that sometimes symbolic representations have a lot of structure. Any symbolic analysis generates a sequence of symbolic representations. If we could examine a sequence of structures and detect a pattern, maybe we can also generalise that pattern. In other words, can I look at a finite history of the analysis, extract some structural pattern and extrapolate from there?

That is what we will try to do today.

### **Slide 10: Scoop of the Day**

The ideas I will present are quite general, but to make them more tangible, I will work within a specific setting.

The abstraction I will consider is the set of regular languages and the symbolic representation is finite automata. For people familiar with alternating automata, I will make some comments at some points.

I will show you a framework for designing extrapolation operators for automata. But design alone is not enough. I will also cover a few basic notions and results about these operators. I also claim that my extrapolation framework is quite general and will give you a few examples to support this claim.

### **Slide 11-13: Extrapolation in Action**

Let's start with a concrete example. Here I have a program with a variable  $x$ . In the loop, if  $x$  is odd, I double it. If  $x$  is even, I double it and add 1.

Suppose I want to determine if this assertion holds. Suppose that the exit condition for the loop is complicated, involving say function calls and pointer dereferences, so we will just abstract it away.

I have an iterative analysis that goes through the loop and collects the values of  $x$ . First, let's abstract the values. We can write these numbers as binary strings. I will write lambda for 0 and all other numbers in the standard way. Because I only have a finite set which is regular, I have not lost any information by doing the abstraction.

Let's bring in the symbolic representation. In each step, the automaton I have here accepts the possible values of  $x$  in the iterations so far. This sequence of automata may be infinite.

This is where extrapolation techniques come in. There are many techniques in the literature, but it is not always clear how they work and what kind of effect they have. Actually, I often have the feeling that there is magic going on. Let's assume that some magic takes place and we get this automaton here.

Is this a good extrapolation? Well, up here it appears that we have an alternating sequence of 0's and 1's. That is exactly the language of this automaton. We can now verify that 511 in binary is a sequence of all 1s, so it will not be in this language and the assertion will not fail.

The rest of my talk is all about how to get from here (finite sequence of automata) to here (limit). The questions are: Can I do this systematically? Is there a good mathematical basis for doing this? Can I prove some interesting properties about the methods I use?

#### **Slide 14: Framework I: Identify Pattern**

The first step is to identify a pattern. This is actually the hardest step because a pattern is a very amorphous notion. What we can do is adopt heuristics. What properties of automata lead us to believe that they are part of a common pattern?

One example is that if two automata have the same suffixes, they might be part of the same pattern. But this is too strong. We could say if they have the same suffixes of a certain length. I will write that like this. Up to length  $k$ , the two automata have the same suffixes.

We could adopt a similar reasoning with prefixes. I could say, it appears that two states are part of the same pattern if they have a common prefix. Or I could even bound the length of the prefix. Say I was looking at prefixes of length 1.

In general, I can formalise heuristic intuition about what constitutes a pattern using a relation. I can define a function, called a relational template, which given two automata computes this relation. That's the first step.

#### **Slide 15-17: Framework II: Collate Similarities**

Now say we are able to identify a pattern between two automata. I want to know which states in one automaton are part of the same pattern. I want to partition the states of the automaton I have, so that each partition contains only related states.

How do I do that? I have a relation, if I take its inverse and compose it with itself, I have a relation over one automaton. There are two problems. If I have something like this: (transitive relation) then I would also like the first and the last states to be related. The other problem is that there may be states constituting a pattern which has not yet emerged in the first automaton. Such a state will not be covered by the relation.

What I do to address these problems is first take the transitive closure and then throw in the identity. Now we have an equivalence partition of the states of  $B$ . In fact, it is the smallest equivalence relation that can be generated from such a relation. Given a relational template and two automata, I can construct a relation and then an equivalence relation. I call this an equivalence template.

#### **Slide 18: Framework III: Generalise Pattern**

A relational template identifies a pattern between two automata. An equivalence template can be used to collect similar states in one automaton. We still need to generalise the pattern we have observed. For this, I will just apply the quotient construction. If two states are in the same partition, merge them.

Quotienting is usually used to minimize deterministic automata, but you can also use it to increase the language of the automaton. Actually, if you take equality of suffixes to be your relational template, then, identifying the equivalence classes and merging them is the same as minimizing the automaton.

The combination of these three elements gives me an extrapolation operator. We have a relational template, construct a relation, then an equivalence partition and finally the quotient.

I hope this appears simple to you. My claim is that extrapolation in this form, is quite widespread and not just in program analysis and verification. Let's look at a few examples.

#### **Slide 19: Specification/Error Mining**

Now let's consider an example from software engineering. Say we want a specification or formal model of a certain API but none is available. The problem is also that we may not have the source code for the server side. One possibility is to examine the client side code and try to obtain a model for API usage.

That's precisely what the authors do in this paper from ISSTA 2007. Here is an example of client side code using the JAVA socket API. The first phase is called data collection. One constructs traces describing API usage by examining the code that is available. This is also the abstraction step – because you are abstracting away from other behaviour in the program. These sets of traces are represented symbolically by a sequence of automata.

Finally, they do some generalisation. In this case, they say that if two states have a common suffix of some length *and* a common prefix of some length, they are related. They then partition the automata and merge these states.

Here is an example of the result of the analysis. A model of API usage consistent with this code: First you configure the socket and then connect. You wait for a response, and get frustrated and close the connection, or you eventually get a response, read and write to the socket and then close it.

## Slide 20: Inductive Inference from Positive Data

I'll start with a subfield of learning theory called inductive inference. In 1967, Mark Gold wrote the founding paper for this area, and one could argue that it is the first computational learning theory paper. He presented different models for learning formal languages.

In the inductive inference model, a learner receives a sequence of examples drawn from a target language. The learner attempts to generalise these examples to describe the target language. This is called a hypothesis. The learning process ends if the learner's hypothesis stabilizes. If the target language is regular, the method is called regular inference.

The initial paper included some negative results and generated more. If you have a class of languages which contains all languages of finitely many strings and at least one language with infinitely many strings, then there exists no inductive inference algorithm. The intuition is that you cannot decide, if you should just wait a few more iterations and have all the strings in the language, or generalise to an infinite language. This means that even regular languages cannot be learnt in this model. So people lost interest in inductive inference.

In 1982, Dana Angluin, reignited the interest with her paper 'Inference of Reversible languages.' She showed that there are families of languages which do not contain all finite languages, but some finite and some infinite ones, and these can be learnt from positive data alone.

If you have a minimal deterministic automaton for a language and you reverse it: that is, you flip all the transitions around, mark the initial states as final and the final states as initial, and if the automaton you now get is deterministic, the language it accepts is called reversible. The reversible languages are those whose minimal deterministic automaton is also reverse deterministic. She gave the following inference algorithm for reversible languages. It's not meant to be legible.

This algorithm can be viewed as the following extrapolation operator. If two states have a common suffix, merge them. This also makes it quite easy to characterise what you will learn. If two states have a common suffix, they will be merged. In the resulting automaton, no two states will have common suffixes. If you reverse the automaton, no two states have common prefixes. So if your learning algorithm was correct, then the target language must be reversible.

## Slide 21: Verification is at the party too

The third example I will give you is from the verification literature. Say you have a system whose behaviour can be modelled using Presburger arithmetic. Finite automata over finite words can be used as a symbolic representation for model checking such systems. That is what Bartzis and Bultan do in their CAV 2004 paper. They design a widening operator because the fixed point computation may not terminate.

Let's see how it works. Two states are related if they have a common prefix, or if they have the same suffix. For example, the state  $p_0$  as a common prefix with  $q_0$  because they are both initial states, and has the same suffixes as  $q_1$ , which you can see because these two sub-graphs here are isomorphic. The state  $q_1$  also has a common prefix with the state  $p_1$ , so these three states ( $q_0, q_1, q_3$ ) will be merged.

The operator in the original paper is actually unsound because they do not use the standard definition and it is not clear how to obtain the equivalence relation given the two relational templates.

Can we say anything else about this operator? Suppose I have a finite set of strings as in the inductive inference setting. I will represent these strings by a suffix tree. That's a tree in which you read the strings backwards (draw example). In a suffix tree, no two nodes have a common suffix. If I apply this operator, I will merge all nodes with common prefixes. So, at the end, no states in the resulting automaton will have a common prefix or a common suffix. If I reverse this automaton, no two states will have a common prefix or suffix either. That means, I can learn those automata whose reversal is also deterministic. So, I can learn reversible languages.

This is one advantage I feel we obtain from such a mathematical framework. We can compare a learning algorithm published in 1982 with a widening operator published in 2004 and say that with the appropriate representation, they can be used to learn the same class of languages.

## Slide 22: YACC for Automata Extrapolation

In total, this is the framework that we can use for constructing extrapolation operators for automata. A single relational template can be used to describe some heuristic observations about how states are related. We saw examples where states are related if they satisfy one condition AND another one, or one condition OR another one. You can construct any Boolean combination of relational templates by taking union, intersection and difference between templates. Each of these is a relational template. From these, we can generate equivalence relations. Again, it is possible to take intersection and unions between equivalence relations.

Given any set of relational templates, we can generate more using such compositions. If you are familiar with alternating automata, the standard quotient can be viewed as taking the disjunction of the transition functions of equivalent states. Instead, we could also take the conjunction of these transition functions – what I call the conjunctive quotient. Then, by merging states, a word is accepted only if all the equivalent states accept that word. Using the conjunctive quotient decreases the language of the automaton.

If you want to over-approximate the limit of an increasing sequence, you can use an extrapolation operator with the standard quotient. If you want to under-approximate the limit of a decreasing sequence, or of a greatest fixed point computation, you can use the conjunctive quotient.

Here is a basic soundness result that the extrapolation operators I have defined with the standard quotient do not decrease the language of an automaton.

## Slide 23: Analysis with Extrapolation

Till now, I have only introduced a framework for designing operators. We also want to apply these operators in some manner to compute an approximation. In this picture, this (the lower ellipse) is the set of pre-fixed points of a function, and this (upper ellipse) is the set of post-fixed points. Here (intersection) we have the fixed points. A standard analysis can be viewed as starting from some initial value  $A_0$  and progressively getting larger. If a least fixed point exists and can be computed in finite time, the sequence will reach here (bottom of upper ellipse).

If we apply an extrapolation operator now and then, using parameters that are given to us, the sequence may progress like this, till we eventually obtain some post-fixed point. That's pretty much what it says here in this formal definition.

By combining the previous soundness statement with induction, we can conclude that the limit of the sequence with extrapolation over-approximates the limit of the original sequence.

This is all very fine, but we don't want an over-approximation in the limit. We want one sometime soon, preferably now!

## Slide 24: Termination

Let's look at how we can design operators that enable the analysis to terminate. Does 'Myhill-Nerode Equivalence' say anything to everyone here? If not, think of this ( $ind(A)$ ) as the number of states in the minimal deterministic automaton for the language of  $A$ .

Here's a characterisation of all extrapolated sequences which terminate. We need that the number of states in the minimal deterministic automaton for the languages in this sequence is bounded infinitely often. Why? Well, the number of minimal automata you can have with a certain number of states is finite, so in this sequence, some language must repeat. From the way I define my sequence, everything beyond that point will be the same and the analysis will terminate.

But this condition itself does not tell us how to obtain termination, because if I apply an extrapolation operator, I have no idea how many states the resulting automaton will have if I minimize and determinize.

There are two conclusions we can draw. If my operator bounds the number of equivalence classes, then surely, the number of states in the minimal deterministic automaton is bounded as well. If I use such an operator infinitely often, my analysis will terminate. This observation is not new. It already exists in the literature. The problem is that most authors have noted that obtaining an operator which computes useful approximations and gives us termination appears to be extremely difficult.

Here's another possibility for ensuring termination. Suppose that if I have some bound on the first argument to the operator, I can conclude that the number of equivalence classes is bounded, then, by

choosing the parameter and operator, I can obtain termination. This may work better than the first case, because we have two knobs to play with to ensure that the sequence terminates, rather than a very strong restriction on the operator itself.

### Slide 25: Termination: Examples

Let's look at some examples. Say the transition relation of both automata here is complete. That means every for every string, there is some path in this automaton. Suppose I relate states which have a common prefix.

Since every string corresponds to a path in each automaton, every state here (second automaton) is related to some state here (first automaton). This means, the number of equivalence classes is bounded by the number of states in this (first) automaton.

Great. So, we know how to design operators which give us termination. If I have two operators that give me termination and I compose them, will I still be able to terminate? The other case is more interesting. If I have two operators that do not give me a termination guarantee and I compose them, is it possible that I now terminate?

### Slide 26-28:

Suppose I take the intersection of two equivalence templates. If both equivalence templates bound the number of equivalence classes I have, then the number of classes in the intersection is also bounded.

What about the union? If two equivalence templates bound the number of equivalence classes, the union can only decrease the overall number. But what if neither bounds the number of equivalence classes? Is it still possible to get a bound?

(slide 20) The answer is yes. Suppose the bound I want to achieve is  $k$ . Let this be the partition I obtain on the states of the automaton. If there are more than  $k$  equivalence classes, then I need to make them disappear somehow. How can we do that? Well, we have another equivalence relation. If a state in the equivalence class  $k + 1$  is related to some state in classes 1 to  $k$  in the second equivalence relation, then this equivalence class will be merged away in the union. We need to be careful that it is related to a state in the classes 1 to  $k$  because if I related all the states in the classes from  $k + 1$  to  $n$  using the second equivalence relation, I would still get  $k + 1$  equivalence classes in total.

(slide 21) That's essentially what these three conditions say. It's also a full characterisation of when you can obtain a bound under union of equivalence relations.

Well, that was termination. I will not just look at one or two other observations that we can naturally make using such a framework.

### Slide 29: Does Representation Matter

One question we can ask is whether it matters what kind of automata we use. Does it have to be deterministic? Does it have to be minimal? The prevailing wisdom in automata based verification is that one should use minimal deterministic automata. It is easy to see where this bit of folklore comes from. The minimal automata are relatively small, hence more space efficient than other deterministic representations. They are also canonical, so operations like detecting a fixed point are efficient.

Here I have two examples with different automata accepting the same language. Say the extrapolation operator merges states which have a common prefix of length 1. The non-minimal DFA here appears to contain a certain pattern. You accept a 1 or any number of 0s or any number of 0s followed by a 1. If we apply the operator to the tree shaped automaton here, we obtain a result that appears to generalise this notion. It accepts any number of 0s followed by a 1.

What if we apply it to the minimal DFA? The states  $s_1$  and  $s_2$  both have 0 as a prefix of length 1 and will be merged. We get an automaton which accepts all strings except the empty string, which is not as good an approximation as the previous one.

There are different possible explanations. One that may occur is that the first automaton has more states than the minimal one, hence more structure and so it is possible to extrapolate better. But the number of states is not really a good measure.

I have another example. The NFA here and the minimal DFA both have 3 states. Let's using this operator and merge states with a common suffix. No two states in the NFA have a common suffix, so the

result of extrapolation is the same automaton. What about the DFA? The state  $t_2$  and  $t_1$  accept 1, so they will be merged. But  $t_1$  and  $t_0$  accept the string 01, so they will be merged as well and we get this result. Not quite the best approximation.

I want to emphasise this. the representation we use can make a big difference for the kind of approximation we obtain. If we do not pay attention to representation related issues, the result of extrapolation may adversely affect the analysis. This also provides us with another possibility. We can try to design operators that are not sensitive to how a regular language is represented.

### Slide 30: Refinement of Operators

I will briefly give you one more short result before I summarise the talk. Refinement is a fairly crucial notion is verification. If I can refine operators and the analysis with extrapolation did not produce a good approximation, we can do better by refining the operator and repeating the analysis.

This diagram shows how to obtain different refinements. If I have two relational templates, taking the intersection of the resulting equivalence classes gives me a more precise operator. Taking the intersection of the relational templates themselves gives me an even more precise operator. The behaviour of operators is symmetric for union.

That brings me to the end of the technical material I have.

### Slide 31: To sum it all up

I started by listing what I believe are three fundamental ideas in program analysis. We need abstraction to cope with undecidability and computational infeasibility of program analysis. Abstraction, within the framework of abstract interpretation has a long history – they had a 30 year anniversary celebration at POPL this year. Various notions such as duality, refinement and completeness have been studied in depth.

Abstraction is not enough because the analysis is performed on a machine and the way in which we represent these abstract facts affects performance. Symbolic representations help us to cope with combinatorial explosion and in the case of infinite sets, are even a necessity.

I also noted that finite abstractions are not enough and infinite abstractions are in general more expressive. One of the tools we use most commonly to cope with infinite objects is induction. The problem is that in this setting we don't have an induction hypothesis. This is why we need extrapolation. To examine the symbolic representation and try to guess the limit of the analysis. What is the state of the art in extrapolation?

### Slide 33: One for the road

Today, I have tried to make the case for extrapolation. I believe it is a concept that should be studied and analysed on par with abstraction. Extrapolation operators are quite rare in the literature. When they exist, it is often clear what effect they have and how this will affect an analysis. That is part of the reason much program analysis and verification maintains a safe distance from extrapolation operators.

I hope what you have seen today provides some motivation to believe that this need not be the case. We can design good operators in a rigorous mathematical framework and analyse their behaviour and properties. In the process, we can discover how these notions are related to similar ideas in other areas of computer science such as learning and hopefully our results will be contributions to those areas as well.

But proving theorems alone should not be enough. I hope that the next time I talk in front of you, I will be able to tell you what tool I have developed and what interesting properties I could prove.

Thank you.