Critical Analysis of Computer Science Methodology: Theory

Björn Lisper
Dept. of Computer Science and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

March 3, 2004

Introduction

Famous quote: “There’s nothing as useful as a good theory”

However, some caution is advisable. Some relevant questions:

- Is the theory accurate enough?
- Is the theory powerful enough to describe anything interesting?
- Can the theory be used for other than toy systems?

We will use *formal verification of system properties* to exemplify

Is the Theory Accurate Enough?

A theory is always limited by its axioms

If the axioms are not in full accordance with the system being modeled, then there will be aspects of the system that the theory will not describe

Such aspects may affect the practical functionality of the system

In reality, any theory has to be approximative to some extent (or it would be too complex to be useful anyway)

Thus, formal proofs of system properties should be viewed with some precaution

Example: Programming Language Semantics

A programming language might be given a *formal semantics*

Proofs of properties of programs, written in the language, can then be carried out

For instance, a program can be proven *correct* w.r.t. some crucial property

Naïvely interpreted, such a program cannot go wrong

However, there are often aspects that the semantics does not cover

A proof of correctness cannot guarantee absence of errors for aspects not covered by the semantics

For instance, semantics for functions often assume that recursive calls can be made to arbitrary depth

This would require an unbounded memory size to store the call stack

But a computer only has a finite, limited memory!

As a consequence, some programs may run out of space even though the semantics says they are correct

Particularly nasty for *reactive* systems (servers, embedded control systems, . . .), which are supposed to run forever

Is the Theory Powerful Enough?

Some theories allow exact reasoning *and* automatic proof methods

In such a theory, everything can be proved by an automatic tool!

An example is *model checking*:

Model checking works over some *finite state model* describing a system, and a *logical formula* specifying some property of the system

If the formula is true, then the system has the property specified by the formula

A proof method can then automatically prove or disprove that the system has the property

The finiteness of the state space is what makes an automatic proof method possible

Model checking typically works by a (more or less clever) traversal of the finite state space of the system, thus checking the formula for each possible state

However, many interesting systems are not finite-state!

Many, even quite simple, software programs are not, for instance

Verification of properties for *infinite-state systems* is typically *undecidable*

Decidable logics, which allow automated proofs, tend to have limited expressiveness. One must be aware of this limitation

This is not to say that model checking is not a useful technique!

Also, there is active research extending model checking to certain kinds of infinite-state systems

Can the Theory be Used for Other than Toy Systems?

Some methods are perfect in principle

However, when applied in practice, it may turn out that the method is not efficient enough to work for other than small toy systems

Model checking is sometimes suffering from this problem

Due to the number of states for the system, which may be large even if finite

For instance, a chip that can store 10^6 bits will have 2^{10^6} possible states

This large number makes a naïve, exhaustive state exploration impossible

Some model checking algorithms use smart representations and algorithms to deal with whole sets of states simultaneously (active research topic), but it’s definitely a problem in practice

Exact Methods vs. Approximate Methods

Model checking is an *exact* method

Answers “yes” if the property is true and “no” if it is false

To some extent, this is the root of its limitations

Sometimes, *approximate* methods are of interest

Such a method may answer “**yes**” (meaning “surely yes”), “**no**” (meaning “surely no”), or “**don’t know**” (meaning “either yes or no, but I cannot tell for sure which”)

The “don’t know” option makes it possible to design approximate methods for undecidable problems

This makes them potentially more widely applicable

It is also possible to have approximate methods give more detailed answers than yes/no, like a range of possible values for a numeric entity

Example: Abstract Interpretation

Abstract interpretation is a theoretical framework for program analysis

It uses *abstract domains* to represent properties of entities

For every entity e of interest in the program, abstract interpretation defines an abstract entity $\#e$ in the abstract domain

This means that e surely has the property represented by $\#e$

Example: an abstract domain of integer pairs (m, n) representing intervals $[m, n]$, and, for each program variable x and program point P, $\#x = (m, n)$ the property "the value of x in P is always in the interval $[m, n]$":

```
int a[17]
...
P: a[i] = ....
```

If the analysis finds, in program point `P`, that $\#\texttt{i} = (a, b)$ where $0 \leq a$ and $b < 17$, then we know we will surely not access the array out of bounds

This property can be quite important for the correctness of the program

Potential Pitfalls of Abstract Interpretation

An abstract domain always contains a “top element” $\top$

This element represents “don’t know”

So if the analysis says that $\#\texttt{i} = \top$, then we know nothing about the value of `i` in the program point in question

Pitfall 1: the analysis returns $\top$ in cases when we could have better knowledge

(Actually, an abstract interpretation always returning $\top$ is correct, however not very useful)

Pitfall 2: the analysis might yield grossly overapproximated answers

For instance, an abstract interpretation based on intervals must return $(1, 10^{17})$ (representing 10^{17} possible values) when the entity can only assume either the value 1 or 10^{17} (two possible values)

This pitfall is likely if the abstract domain has overly simplistic values

(Pitfall 1 is a special case of pitfall 2)

Pitfall 3: the analysis might be very resource-consuming

Especially true if it uses an abstract domain with more elaborate property representations

Such representations can allow a more precise analysis. So there is a tradeoff between precision and speed/memory requirements

In conclusion, approximative methods can also suffer from similar problems as exact methods (although they can be used for a wider class of problems)

Conclusion

A theory can never model all aspects of a system

A proof about the system is only valid for the aspects actually modeled by the theory

A sound method based on a sound theory may still have limitations, due to:

- inexpressiveness of the theory
- complexity issues
- impreciseness due to large approximations