

Challenges in Computational Software Engineering

Ali Mili,
College of Computing Science
New Jersey Institute of Technology
Newark NJ 07102-1982
(973)596 5215, mili@cis.njit.edu

Frederick Sheldon
U.S. DOE Oak Ridge National Lab
PO Box 2008, MS 6085
Oak Ridge TN 37831-6085
(865)576 1339, sheldonft@ornl.gov

1 Next Generation Software

Broadly speaking, it is possible to characterize next generation software by the following features: size; complexity; distribution; heterogeneity; etc. In the face of such complexity, it is natural to turn to the tool of choice that scholars have always used to maintain intellectual control, viz mathematics; yet, paradoxically, mathematics has remained of limited use in dealing with software engineering in the large.

Automated tools, built on computational models of software engineering, are required to help fill the wide gap between human capabilities and the daunting task of designing, analyzing, and evolving modern software systems [1]. In this position paper, we briefly discuss some computational issues and some automation issues pertaining to this gap.

2 Prospects in Program Analysis: Functional Properties

Despite several decades of research and development, the automated analysis of software artifacts remains an open challenge. While there is an abundance of CASE tools, that assist software engineers in the analysis and synthesis of software systems, these are usually fairly superficial, in the sense that they are incapable of dealing with the detailed semantics of such systems. Tools that are capable of capturing *all* the functional properties of software artifacts are still needed. Some of the research issues that stand in the way of deriving such tools include:

- *Capturing Architectural Information.* In the early days of software engineering, most software products were data processing applications, whose behavior can be readily modeled by a function (mapping an input file to an out-

put file, mapping an input screen to an output screen, etc). Modern applications do not fit this simple model, and typically include complex temporal interactions, distributed state information, complex combinations of events, and as a consequence, non trivial external behavior. In addition to capturing the function of software components, we must also capture attributes of the architecture, and their impact on the behavior of the system.

- *Program Functions Slicing.* Programs are typically very information-rich, and trying to derive their function may require some divide-and-conquer strategies. One possible way to derive the function of a program in a stepwise manner is to extract successive specifications R_i , $i = 1, 2, 3, \dots$, such that

$$[P] \sqsupseteq R_i,$$

where \sqsupseteq represents the refinement ordering, then infer (by lattice theory)

$$f \sqsupseteq R_1 \sqcup R_2 \sqcup R_3 \sqcup \dots \sqcup R_n,$$

where \sqcup is the join in the refinement lattice. Because R_i 's can be arbitrarily weak (non-refined), the statements that $[P]$ refines R_i can be arbitrarily easy to establish. It is possible to prove, under some conditions, that if the join of the R_i is deterministic, then

$$f = R_1 \sqcup R_2 \sqcup R_3 \sqcup \dots \sqcup R_n.$$

We may refer to this method as *program function slicing*, by contrast with the well known technique of *program slicing*. The proposed technique slices the program function (by deriving one component of it at a time) rather than to slice the program (one statement sequence at a time).

- *Combining Data Abstraction and Control Abstraction.* A complete, exhaustive analysis of program functions requires that we model the program's data structures and control structures. In practice, these two models have seldom been used together: Approaches that focus on program function usually assume simple data types whose axiomatization is straightforward. On the other hand, approaches that focus on data modeling usually do not model program functions above the *method* level. For a comprehensive analysis, we must combine data models and control models, and explore means to combine their respective axiomatizations.
- *Functional Analysis as Language Processing.* Language processing (compiler construction, syntax analysis, code generation) has been one of the most successful areas of domain-specific automated software engineering. One can produce a compiler by submitting to a *compiler generator* the BNF of the target language as well as a semantic definition that accompany each BNF production. This creates a strong incentive for us to model program analysis as language processing. In the context of the divide and conquer strategy advocated above, we can model the analysis activity by means of attribute grammars, using two attributes: A *synthesized attribute*, which represents the function of programs or program parts; and an *inherited attribute*, which represents the specification against which we wish to prove refinement.

For example, we can consider that each statement of our programming language has two attributes: a function (*func*) and a specification (*spec*). Given the following BNF rule

$$S ::= S1; S2$$

we produce the following semantic rule for the synthesized attribute:

$$S.func := S1.func \circ S2.func,$$

and the following semantic rules for the inherited attribute:

$$S1.spec := S.spec \setminus S2.func,$$

$$S2.spec := S.spec // S1.func.$$

The synthesized attribute propagates the program function from the leaves of the parse tree

to the root; and the inherited attribute decomposes the specification from the root to the leaves.

- *A Product Line of Functional Analysis Tools.* The biggest surprise one encounters when trying to extract the function of a program is that the extraction of the function is actually not difficult. What is most difficult is to present the extracted function to the user in a way that allows her/ him to relate to it and understand it. The second important realization in this regard is that, in order to present the computed function in a way that is meaningful to the user, we must deploy a great deal of domain-specific knowledge. A well defined paradigm for supporting this kind of development is the paradigm of *product line engineering*, where we build a set of core assets that capture the requisite program analysis knowledge, then we derive the support structure that allows us to integrate domain specific knowledge.

3 Prospects in Program Analysis: Dependability

Non functional attributes of software systems are attributes that do not deal with the system's services per se, but rather with the conditions under which these services are delivered. An important non-functional attribute of software systems is *dependability*, which is composed of reliability, safety, security and availability. The three first attributes, are known collectively as *Surety*. We advocate to model them in a common framework, according to the following premises:

- All three are modeled symbolically by a refinement property, of the form

$$[P] \sqsupseteq R,$$

for some ordering relation \sqsupseteq and some specification R . The exact definition of the ordering (\sqsupseteq) and the specification (R) depend on the specific property.

- Surety claims and goals can be represented, not as absolute logical statements (as written above), but possibly as probabilistic, conditional statements, of the form

$$\Pi([P] \sqsupseteq R|A) \geq p,$$

i.e. the probability that $[P]$ refines R under assumption A is greater than or equal to p .

- In order to acknowledge the relative gravity of various failure conditions, we associate with each claim of surety a *failure cost*, that quantifies the cost of failing to meet the specified requirement. Hence for example, we could write

$$\nu(\sqsupseteq, R) = C,$$

i.e. the cost of failing to refine (by \sqsupseteq) specification R is C .

- We must develop means to compose surety claims and decompose surety goals, as well as means to quantify individual measures and to certify that we have satisfied a particular surety goal.
- The modeling of the various dimensions of surety must highlight the interdependencies that exist between them (some dimensions depend on others, some imply other, etc).

References

- [1] Alan Hevner, Richard Linger, and Gwendolyn Walton. Next generation software engineering: Transformation to a computational engineering discipline. In *Hawaii International Conference on Systems Sciences*, Lihue, HI, January 2006.