Reliable Software Construction: A Logic Programming Based Methodology *

Gopal Gupta
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083
gupta@utdallas.edu

Abstract

In this position paper we investigate how logic programming technology can aid software development. The overall goal is to provide a framework for specification and verification that is "computational" in nature rather than being based on traditional, more complex formalisms such as theorem proving and term rewriting. Two approaches are discussed. In the first approach, given a program written in a traditional language (e.g., C), an equivalent logic program is automatically obtained. This equivalent logic program serves as a high level abstraction of the original program and can be put to a number of uses including verification, structured debugging and generation of provably correct target code. The second approach is centered around domain specific languages. Given a task for which a software system is to be developed, a high-level domain specific language (DSL) is first designed. Domain experts can use this DSL for writing programs at their level of abstraction. Logic programming provides a framework in which programs written in this DSL can be interpreted, compiled, debugged, verified, and profiled.

1 Introduction

Techniques for developing reliable software and formal methods have been an active topic of research. However, while immense progress has been made, formal methods are still not commonly used by practicing engineers. We believe that this is because of the steep learning curve involved in using these methods in practice. Most of the formal methods require the system to be specified (axiomatized) in some complex logic. The property to be verified has to be specified in this logic as well, and a tool is provided that will automatically show that the property to be verified is entailed by the system specification. Writing this specification requires considerable expertise on part of the user. For example, the user may be required to write the specification and the property to be verified in first order logic (e.g., if a theorem proving system is going to be used for verification), temporal logic (e.g., if a model checker is going to be used), etc. It is unreasonable to assume that ordinary programmers possess such knowledge. Even if they did, writing a specification is a non-trivial task. Developing a specification for a system essentially amounts to writing the axioms that capture the behavior of that system. Knowing that these axioms are correct, or for that matter, knowing if the exact axioms needed—no more and no less—have been included is a difficult task. Additionally, the task of developing the specification for the purpose of verification is separate from the task of implementation.

2 Specification vs. Implementation

We believe that current formal methods place too much burden on the user, and that is why formal methods are not part of a software engineer’s standard arsenal. Let us draw an analogy between the software construction process and building a piece of complex mechanical machinery. A mechanical engineer will perhaps first make some drawings (figure 1) and then translate it to specific instructions (figure 2) to be followed by an assembly line worker. The mechanical engineer can answer questions pertaining to correctness of his design based on the drawing (e.g., in figure 1, he/she can affirm by visual inspection that parts #16681 and #16678 are on the same axis). Note also that the drawing schemata leaves the exact order in which the assembly is to be carried out unspecified: there are several ways in which this assembly can be accomplished; the instructions in figure 2 specify one of them. The question of validation (namely, the instructions in figure 2 are a correct realization of the schemata in 1) is much harder to answer. There are two ways we could validate the instructions: (i) rather than have the mechanical engineer write these instructions, derive them automatically from the pictorial schemata; (ii) given the instructions, automatically derive the pictorial schemata and then confirm that indeed the schemata derived is identical to what the engineer had in mind.

*The author has been partially supported by NSF grants CCR 9900320, CCR 9820852, INT 9904063, HRD 9800209, and EIA 9810732
In terms of software development, the pictorial assembly schemata corresponds to the mental model that the programmer builds in her mind, while the instruction corresponds to the actual code she will write to realize the system. Current formal methods require that the software engineer specify this mental model she has developed in some form of logic, use it to verify properties of interest, and finally develop an implementation separately. Given that most software engineers are not trained to write formal specifications, formal methods hardly get used in practice. We need methods that make use of what a software engineer has been trained to do best—program computations. The rest should be accomplished automatically, once the computational steps have been specified.

![Assembly Schemata](image)

**Figure 1: Assembly Schemata**

1. Mount the wire-pin (#13840) on the slot in the machine.
2. Insert coil spring (#16682) on side-cylinder in #16677.
3. Align part #16677 with the machine and put the cylindrical pin (#16697) through the hole.
4. Insert the small cylindrical pin (#16680) in the part #16677. Slide the rod (#16679) through the slot.
5. Screw nut (#17352) on the rod.
6. Insert the slotted cylinder (#16678) on the top of part #16677, aligning the protruding key with the slot.
7. Insert part #16681 in part #16678.

**Figure 2: Imperative Instructions**

### 3 A Logic Programming-based Solution

In this paper we propose an approach in which a logic program is viewed as an analog of the “assembly schemata” while the “instructions” correspond to an implementation developed in an imperative language (e.g., C). Given a specification written as a logic program, an engineer can query it to check that indeed the implementation captures his/her mental model (just as a mechanical engineer would visually inspect the schemata to check for correctness). However, we do not require that programmers write their specification in a logic programming language. Rather we espouse the approach that domain specific languages (DSL) must be developed for this purpose. DSLs are specialized languages designed for solving problems in a particular domain [3, 4]. DSLs allow a domain expert, who may not be an expert programmer, to program at their level of abstraction. DSL programs should be thought of as specifications (high-level schemata) and less as programs. To enable the user to verify their designs, we propose that Horn logical denotational semantics [1] of the DSL should be developed once a DSL has been defined. The semantics can be used to obtain denotation of a DSL program. This denotation—a logic program—can be thought of as the assembly schemata and used to verify the correctness of the design [1]. The logical denotational semantics, coupled with partial evaluation, can also be used to derive efficient implementations of the DSL. Note that the process of deriving more efficient code (compilation) is provably correct, so no validation is needed. Details can be found elsewhere [2, 4].

If a domain specific language cannot be developed, then we propose an approach in which we try to automatically construct the schemata from the implementation. Most often engineers develop an informal design and go straight to an implementation (i.e., to developing computer code). This implementation can be used to automatically derive the schemata, which the software writer can then experiment with to confirm that indeed the design that his implementation encodes is correct. This can also be accomplished through the use of Horn logic encoded denotational semantics [1, 5]. Suppose the implementation is coded in C. Then Horn logic denotational semantics of C can be developed, and used to obtain a denotation of the implementation [5]. The denotation is a logic program, which can be used for verification purposes. Details can be found elsewhere [2].

### References


