Rapidly Prototyping Implementation Infrastructure of Domain Specific Languages: A Semantics-based Approach*

Abstract

Domain Specific Languages (DSLs) are high level languages designed for solving problems in a particular domain, and have been suggested as means for developing reliable software systems. However, designing of a domain specific language is a difficult task. The design of a domain specific language will evolve as it is used more and more and experienced is gained by its designers. Being able to rapidly develop the implementation infrastructure (interpreter, compiler, debugger, profiler, etc.) of a domain specific language is thus of utmost importance so that as the language evolves, the implementation infrastructure can keep pace. In this paper we present a framework for automatically generating interpreters, compilers, debuggers, and profilers from semantic specification of a domain specific language. We illustrate our approach via the SCR language, a language used by the US defense department for developing control systems.

1 Introduction

Writing software that is robust and reliable is a major problem that software developers and designers face today. Development of techniques for building reliable software has been an area of study for quite some time. Recently, approaches based on *domain specific languages* (DSL) have been proposed [1, 3, 2, 4, 5]. In the DSL approach, a domain specific language is developed to allow users to solve problems in a particular application area. A DSL allows users to develop complete application programs in a particular domain. Domain specific languages are very high level languages in which domain experts can write programs *at a level of abstraction at which they think and reason.* DSLs are not "general purpose" languages, rather they are supposed to be just expressive enough to "capture the semantics of an application domain" [2]. The fact that users are able to code problems at the level of abstraction at which they think and the level at which they understand the specific application domain results in programs that are more likely to be correct, that are easier to write, understand and reason about, and easier to maintain. As a net result, programmer productivity is considerably improved.

The task of developing a program to solve a specific problem involves two steps. The first step is to devise a solution procedure to solve the problem. This steps requires a domain expert to use his/her domain knowledge, expertise, creativity and mental acumen, to devise a solution to the problem. The second step is to code the solution in some executable notation (such as a computer programming language) to obtain a program that can then be run on a computer to solve the problem. In the second step the user is required to map the *steps* of the solution procedure to *constructs* of the programming language being used for coding. Both steps are cognitively challenging and require considerable amount of thinking and mental activity. The more we can reduce the amount of mental activity involved in both steps (e.g., via automation), the more reliable the process of program construction will be. Not much can be done about the first step as far as reducing the amount of mental activity is involved, however, a lot can be done for the second step. The amount of mental effort the programmer has to put in the second step depends on the "semantic" gap between the level of abstraction at which the solution procedure has been conceived and the various constructs of the programming language being used. Domain experts usually think at a very high level of abstraction while designing the solution procedure. As a result, the more low-level is the programming language, the wider the semantic gap, and the harder the user's task. In contrast, if we had a language that was right at the level of abstraction at which the user thinks, the task of constructing the program would be much easier. A domain specific language indeed makes this possible.

A considerable amount of infrastructure is needed to support a DSL. First of all the DSL should be manually designed. The design of the language will require the inputs of both computer scientists and

*The authors have been partially supported by grants from the NSF, the Department of Education and Environment Protection Agency.

domain experts. Once the DSL has been designed, we need its implementation infrastructure, i.e., a program development environment (an interpreter, a compiler, debuggers, profilers, editors, etc.), to facilitate the development of programs written in this DSL. Observe that the time taken to initially design the DSL is dependent on how rapidly the DSL can be implemented, since the ability to rapidly implement the language allows its designers to quickly experiment with various language constructs and with their various possible semantics.

Note also that like any other language a domain specific language will evolve as it is used more and more to write programs. Users and language designers gain experience with use of a DSL as more and more programs are written. This inevitably results in language features being modified or added to the DSL. If the DSL changes, then obviously its implementation infrastructure must change as well. The success of the domain specific language methodology will depend on how rapidly this implementation infrastructure can be modified to reflect the changes in the language. Experience has shown that the time for a domain specific language to mature can be several years [4]. A significant part of this time is spent developing and modifying the implementation infrastructure as the DSL changes. Clearly, if the implementation infrastructure can be rapidly developed/modified, then the time for the DSL to mature will be reduced, making a DSL based approach to software engineering practical.

In this paper we show how a semantics based framework based on (constraint) logic programming can be used for rapidly developing and modifying interpreters/compilers as well as debuggers and profilers for DSLs. In this framework, the syntax and semantics of the DSL are expressed using Horn logic. The Horn logic coded syntax and semantics is executable, automatically yielding an interpreter. Given this semantics-based interpreter for the DSL and a program written in this DSL, the interpreter can be *partially evaluated* [27] w.r.t. the DSL program (e.g., using a partial evaluator for Prolog such as Mixtus [40]) to automatically generate compiled code. The semantic specification can be extended with hooks to automatically produce a debugger/profiler for the language. Given that the interpreter, compiler, and the debugger are all obtained automatically from the syntax and semantic specification, the process of developing the infrastructure for supporting the DSL is very rapid. Also, as the DSL changes, its formal semantic specification can be modified to reflect the changes, and then an interpreter, compiler, debugger, or a profiler automatically derived from this semantic specification.

An obvious candidate framework for specifying the semantics of a domain specific language is denotational semantics [41] However, the denotational semantics we use has an operational flavor; it's denotational in that it is declarative. Denotational semantics has three components: (i) syntax, which is typically specified using a BNF, (ii) semantic algebras, or value spaces, in terms of which the meaning is given, and, (iii) valuation functions, which map abstract syntax to semantic algebras. In traditional denotational definitions, syntax is specified using BNF, and the semantic algebra and valuation functions using λ-calculus. By using Horn Logic (or pure Prolog) instead we can specify the syntax as well in an executable fashion (i.e., the syntax specification as a *definite clause grammar* automatically yields a parser). Switching to Horn logic also facilitates development of semantics based verification tools.

We assume in this paper that the reader is familiar with denotational semantics, logic programming, and partial evaluation.

2 Horn Logical Semantics of Languages

Traditional denotational definitions express syntax as BNF grammars, and the semantic algebras and valuation functions using λ-calculus. Instead, we use Horn-clause Logic (or pure Prolog) and constraints to code all the components of the denotational semantics of programming languages [15]. There are three major advantages of using Horn clauses and constraints for coding denotational semantics.

First, a Parser can be obtained from the syntax specification with negligible effort. The BNF specification of a language $\mathcal{L}$ can be trivially translated to a *Definite Clause Grammar* (DCG) [43]. As a result, the syntax specification can be loaded in a Prolog system, and a parser automatically obtained. This parser can be used to parse programs written in $\mathcal{L}$ and obtain their parse trees (or syntax trees). Thus, the syntactic BNF specification of a language is easily turned into *executable syntax* (i.e., a parser).

Second, the semantic algebra and valuation functions of $\mathcal{L}$ can also be coded in Horn-clause Logic. Since Horn-clause Logic or pure Prolog is a declarative programming notation, just like the λ-calculus, the mathematical properties of denotational semantics are preserved. This is because logic programming is a declarative programming paradigm based on relations, and relations subsume functions (the

```
Program ::= C.
C ::= C1;C2 |
   loop while B C endloop while |
   if B then C1 else C2 endif |
   I := E
E ::= N | Identifier | E1 + E2 |
   E1 - E2 | E1 * E2 | (E)
N ::= 0 | 1 | 2 | ...  | 9
Identifier ::= w | x | y | z
```

Figure 1: BNF grammar

basis of λ-calculus). Since both the syntax and semantic part of the denotational specification are expressed as logic programs, they are both executable. These syntax and semantic specifications can be loaded in a logic programming system and executed, given a program written in $\mathcal{L}$. This provides us with an interpreter for the language $\mathcal{L}$. In other words, the *denotation*[1] of a program written in $\mathcal{L}$ is executable. This executable denotation can also be used for many applications, including automated generation of compiled code.

Finally, Horn-logical semantics can be used for automatic verification and consistency checking. The Horn logical denotation of a program is a collection of Horn clauses. Since Horn clauses are logical formulas, the denotation of the program ($\mathcal{D}_p$) can be regarded as an axiomatization of the solution expressed in that program. To verify a property ϕ, we just need to show that ϕ is a logical consequence of $\mathcal{D}_p$. That is, $\mathcal{D}_p \models \phi$. The relational nature of logic programming allows for the state space of a program written in $\mathcal{L}$ to be explored with ease, thus if the property ϕ is expressible as a Horn clause and $\mathcal{D}_p$ is finite, then a logic programming engine can be employed for performing verification [26, 15, 17].

Next, we give an illustrative example. Consider a very simple subset of a Pascal-like language that contains an assignment, if-then-else, and while-do statements. To keep matters simple, assume that the only possible variable names allowed in the program are `w, x, y` and `z`, and that the only data-type allowed is integer. Assume, again for simplicity, that constants appearing in the program are only one digit long. The context free grammar of this language is given in Figure 1. This BNF is easily transformed into a definite clause grammar (DCG) by a simple change in syntax [43] (plus removal of left-recursion, if a Prolog system is going to be used for its execution). This DCG is shown in Figure 2.

To illustrate the syntax rules, consider the rule for the while loop. The rule (in Figure 2) states that

[1]We refer to the denotation of a program under the Horn-logical semantics as its *Horn logical denotation.*

```
SYNTAX:
   program(p(X)) --> comm(X), [.].
   comm(X) --> comm1(X).
   comm(comb(X,Y)) --> comm1(X),[;],comm(Y).
   comm1(assign(I,E)) --> id(I),[:=],expn(E).
   comm1(ce(X,Y,Z)) -->
             [if],expn(X),[then], comm(Y),
                  [else], comm(Z), [endif].
   comm1(while(B,C)) -->
             [loop, while], bool(B),
                 comm(C), [endloop, while].
   expn1(id(X)) --> id(X).
   expn1(num(X)) --> n(X).
   expn1(e(X)) --> ['('],expn(X),[')'].
   expn(X) --> expn1(X).
   expn(add(X,Y)) --> expn1(X),[+],expn(Y).
   expn(sub(X,Y)) --> expn1(X),[-],expn(Y).
   expn(multi(X,Y)) --> expn1(X),[*],expn(Y).
   bool(equal(X,Y)) --> expn(X),[=],expn(Y).
   bool(great(X,Y)) --> expn(X),[>],expn(Y).
   bool(less(X,Y)) --> expn(X),[<],expn(Y).
   id(x) --> [w].       id(x) --> [x].
   id(y) --> [y].       id(z) --> [z].
   n(0) --> [0].
   ....
   n(9) --> [9].
```

Figure 2: DCG for the BNF

the while statement consists of the keyword `loop while` (enclosed in square brackets), followed by a boolean expression, followed by the body of the loop, and then the keywords `endloop while;`. The syntax rule also (recursively) synthesizes the parse tree during parsing: the Prolog term `while(B,C)` is created, where `B` is the parse tree that corresponds to the boolean expression (and is produced by the rule `bool(B)` that parses such expression), and `C` is the parse tree that corresponds to the body of the loop (and is produced by the rule `comm(C)` that parses the body).

The DCG is a logic program and, when executed, a parser is automatically obtained. This DCG is shown in Figure 2. This DCG parses a program written in this simple language and produces a parse tree for it. For example, the query to parse the program for computing the value of y^x and placing it in variable z is as follows:

```
?- program(P, [z,=,1,;, w,=,x,;,
            loop, while,w,>,0,
              z,=,z,*,y,;, w,=,w,-,1,
            endloop, while], []).
```

This query will parse the program and verify that it is syntactically correct, and produce the parse tree shown below:

```
P = p(comb(assign(z,num(1)),
   comb(assign(w,id(x)),while(
```

```
    great(id(w),num(0)),
       comb(assign(z,multi(id(z),id(y))),
       assign(w,sub(id(w),num(1))))))))
```

The Horn logical semantics can be defined next, by expressing the semantic algebra and the valuation functions as logic programs. To keep the example simple, we assume that in the semantic definition the input is initially found in variables `x` and `y` and the answer is computed and placed in the variable `z`. The semantic algebra consists simply of the `memory store` domain, realized as an association list of the form `[(Id, Value) ...]` with operations for creating, accessing, and updating the store, and is given below as a logic program:

```
          SEMANTIC ALGEBRA:
initialize_store([]).
access(Id,[(Id,Val)|_ ],Val).
access(Id,[_|R],Val) :-
              access(Id,R,Val).
update(Id,NewV,[],[(Id,NewV)]).
update(Id,NV,[(Id,_)|R],[(Id,NV)|R]).
update(Id,NewV,[P|R],[P|R1]) :-
                 update(Id,NewV,R,R1).
```

Next, the valuation functions that impart meaning to the language are specified, again as logic programs. These valuation functions, or valuation predicates, relate the current memory store, a parse tree pattern whose meaning is to be specified, and the new store that results from executing the program fragment specified by the parse tree pattern. These valuation predicates are shown in Figure 3; the very first valuation predicate takes the two input values, that are placed in the store locations corresponding to `x` and `y`.

To illustrate the semantic rules, consider the valuation predicate for the while loop. The semantic rule takes the parse tree for the `while` loop and a memory store (`Store`) as an argument, and produces the final memory store (`OutStore`) that results after the while loop finishes execution. The rule is defined recursively and is self-explanatory. Similar syntax and semantic rules are given for the other constructs.

Once the syntax, semantic algebras, and valuation functions are defined as a logic program, an interpreter is immediately obtained. To execute the exponentiation program on this interpreter we need to put the syntax and semantics specification together:

```
main(ValX, ValY, ValZ) :-
    program(P, [z,=,1,;, w,=,x,;,
                loop, while,w,>,0,
                     z,=,z, *, y, ;,
                     w,=,w,-,1,
                endloop,while], []),
    prog_eval(P,ValX,ValY,ValZ).
```

The predicate `main` is the *denotation* of the exponentiation program.

If the above syntax and semantics rules are loaded in a logic programming system and the query `?- main(5,2,ValZ)` for computing the value of 2^5 is posed, then the value of `ValZ` will be computed as `32`. Notice that switching to logic programming for specifying denotational semantics results in a complete interpreter. Thus, Horn logical semantics allows us to obtain the interpreter for a language from its semantic definition very rapidly.

3 Provably Correct Compilation

We next show how compiled code can be automatically generated from this interpreter via partial evaluation. The theory of automatic generation of compiled code, a compiler and a compiler generator for a language $\mathcal{L}$ from the definition of its interpreter is well studied in the field of programming languages [12, 27]:

- Given an interpreter for language $\mathcal{L}$, and a program P, during the interpretation process P can be regarded as static input to the interpreter, while any input supplied to program P while it is running is the dynamic input. Thus, the interpreter can be partially evaluated w.r.t. P. Partially evaluating the interpreter for $\mathcal{L}$ w.r.t. P automatically yields *compiled code* for P. This is known as the first Futamura projection [27]. We assume that the partial evaluator is a program that takes as input the interpreter for $\mathcal{L}$ and the program P written in $\mathcal{L}$.

- Since the partial evaluator is a program itself, we can treat the interpreter as a static input of the partial evaluator. The partial evaluator itself can be partially evaluated w.r.t. the interpreter; the resulting program is a compiler of $\mathcal{L}$. This is known as the second Futamura projection [27].

- Finally, the partial evaluator can be partially evaluated w.r.t. itself. This yields a program that given the interpreter for $\mathcal{L}$ automatically yields a compiler for $\mathcal{L}$. The resulting program is thus a compiler generator. This is known as the third Futamura projection [27].

In this paper, we are only interested in the First Futamura projection. We have already obtained an

```
prog_eval(p(Comm), Val_x, Val_y, Output) :- initialize_store(Store),
      update(x, Val_x, Store, Mst), update(y, Val_y, Mst, Nst),
      comm(Comm, Nst, Pst), access(z, Pst, Output).
comm(comb(C1, C2), Store, Outstore) :- comm(C1, Store, Nstore),
                          comm(C2, Nstore, Outstore).
comm(while(B, C), Store, Outstore) :-
        (bool(B, Store) -> comm(C, Store, Nstore),
           comm(while(B, C), Nstore, Outstore); Outstore=Store).
comm(ce(B, C1, C2), Store, Outstore) :- (bool(B, Store) ->
           comm(C1, Store, Outstore); comm(C2, Store, Outstore)).
comm(assign(I, E), Store, Outstore) :-
       expr(E, Store, Val), update(I, Val, Store, Outstore).
expr(add(E1, E2), Store, Result) :- expr(E1, Store, Val_E1),
       expr(E2, Store, Val_E2), Result is Val_E1+Val_E2.
expr(sub(E1, E2), Store, Result) :- expr(E1, Store, Val_E1),
             expr(E2, Store, Val_E2), Result is Val_E1-Val_E2.
expr(multi(E1, E2), Store, Result) :- expr(E1, Store, Val_E1),
             expr(E2, Store, Val_E2), Result is Val_E1*Val_E2.
expr(id(X), Store, Result) :- access(X, Store, Result).
expr(num(X), _, X).
bool(great(E1, E2), Store) :- expr(E1, Store, Eval1),
            expr(E2, Store, Eval2), Eval1 > Eval2.
bool(less(E1, E2), Store) :- expr(E1, Store, Eval1),
            expr(E2, Store, Eval2), Eval1 < Eval2.
bool(equal(E1, E2), Store) :- expr(E1, Store, Eval),
            expr(E2, Store, Eval).
```

Figure 3: Valuation Predicates

interpreter for the language from its Horn-logical semantics. If we had a partial evaluator for pure Prolog, then we could use it to partially evaluate the interpreter obtained w.r.t. the example program above for computing y^x. Indeed such partial evaluators do exist, the most well known one being the Mixtus system from the Swedish Institute of Computer Science [40]. Mixtus is in fact a partial evaluator for full Prolog. Removing the semantic algebra for the store from our definition, followed by the partial evaluation (using Mixtus) of the interpreter w.r.t. the program for computing y^x, results in compiled code. Essentially, the original denotation of the program is simplified to the point where it only contains calls to the semantic algebra functions.

The program obtained as a result of partial evaluation is shown in Figure 4. In this program, after partial evaluation, a series of memory `access`, memory `update`, arithmetic and comparison operations are left, that correspond to `load`, `store`, arithmetic, and comparison operations of a machine language. The while-loop, whose meaning was expressed using recursion, will (always) partially evaluate to a *tail-recursive* program. These tail-recursive calls are easily converted to iterative structures using jumps. Machine code is thus a few simple transformation steps away. The code generation process is provably correct, since target code is obtained automatically via partial evaluation. Of course, we need to ensure that the partial evaluator works correctly. However, this needs to be done only once. Note that once we prove the correctness of the partial evaluator, compiled code for programs written in *any language* can be generated in a provably correct manner as long as the Horn-logical semantics of the language is given.

Note, that the compiled code generated has the store argument stringing through it. It is relatively easy to eliminate this store argument by using *definite clause grammars* to specify valuation functions as well; details can be found elsewhere [16].

4 Automatic Generation of Debuggers and Profilers

A debugger allows a programmer to view the snapshot of the state at any point in the program. Given that the denotational semantics is a map from parse tree to memory store, the snapshot of the current state can be viewed merely by observing the current store. Thus, an interpreter derived from the semantic specification of a language can easily extended to

```
main(X,Y,A) :-
      initialize_store(B),
      update(a,X,B,C),
      update(b,Y,C,D),
      update(z,1,D,E),
      access(x,E,F),
      update(w,F,E,G),
      commandwhile(G,H),
      access(z,H,A).
commandwhile(A,B) :-
     (access(w,A,C),
      0<C ->
         access(z,A,D),
         access(y,A,E),
         F is D*E,
         update(z,F,A,G),
         access(w,G,H),
         I is H-1,
         update(w,I,G,J),
         commandwhile(J,B)
       ; B=A ).
```

Figure 4: Compiled Code

produce a debugger. A new command,

```
break Condition
```

where `Condition` is optional, is added to the language. The semantics of the `break` command is defined in terms of a predicate that, if the denotation of `Condition` is true, accepts a series of commands from the user, which the user can use to examine the state of the store. At the minimum, a command that allows the user to examine the values of variable in the store, and another command that will cause the predicate to succeed and return the control to the interpreter is needed in this language of commands for debugging.

The semantics of this command language can also be given using the same Horn logical framework, and the interpreter obtained for this language of debugging related command can be seamlessly integrated with the interpreter of the language via the semantic predicate for the `break` command. The `break` command can be thought of as a hook that allows the interpreter to escape to the debug mode.

A profiler can be automatically generated as well. Essentially, the semantics can be extended so that it keeps track of various types of profiling data in a data-structure that is included in the global state. This data-structure can be used to provide an execution profile when one is needed for a program run.

5 Applications

The Horn logical semantics based approach for rapidly implementing DSLs has been applied to a number of practical applications. These include generating code for parallelizing compilers [18] and for controllers specified in Ada [26] (for verification purposes) in a provably correct manner, and rapidly implementing a domain specific language for bioinformatics [36] and most recently generating code in a provably correct manner for the Software Cost Reduction (SCR) framework, discussed next.

5.1 SCR: Software Cost Reduction Method

The SCR (Software Cost Reduction) requirements method is a software development methodology introduced in the 80s [25] for engineering reliable software systems. The target domain for SCR is real-time embedded systems. The SCR method has been extended to describe not only functional requirements (the values the system assigns to outputs) but also nonfunctional (e.g., timing and accuracy) requirements. A number of automatic tools have been developed to aid in formal specification, verification and validation of systems using the SCR method [23]. SCR has been applied to a number of practical systems, including avionics system (the A-7 Operational flight Program [32, 6]), a submarine communication system, and safety-critical components of a nuclear power plant [42]. The SCR method is scalable and has been applied to document requirements of the Lockheed's C-130J Operational Flight Program which resulted in approximately 100,000 lines of Ada code.

The SCR method describes system behavior by a mathematical relation between monitored variables and controlled variables. This relation is concisely specified using *condition*, *events* and *tables*. A condition is a predicate defined on one or more variables in the specification. An event occurs when any variable changes values. The environment changes monitored values and causes input events. In response, the system updates the value of one or more controlled variable according to some relations. Each SCR table specifies the required value of a variable as a mathematical function defined on conditions and events. There are three kinds of table used in SCR: *condition* tables, *event* tables, and *mode transition* tables. The tables facilitate industrial application of the SCR method since engineers find tables relatively easy to understand and to develop [22]. In additional, tables can describe large quantities of requirement information concisely.

5.2 The Four-Variable Model

There are several versions of SCR. One of the most important versions is the Four Variable Model. The Four Variable Model [11], illustrated in Figure 5, represents requirements as a set of mathematical relations on four sets of variables (monitored, controlled, input, and output variables). A monitored variable represents an environmental quantity that influences system behavior, while a controlled variable represents an environmental quantity the system controls. A black box specification of required behavior is given as two relations (REQ and NAT) from the monitored quantities to the controlled quantities (rather than inputs to outputs). NAT, which defines the set of possible values, describes the natural constraints on the system behavior, such as constraints imposed by physical laws and the system environment. REQ defines the additional constraints on the system to be built as relations the system must maintain between the monitored and the controlled quantities [23, 22].

A domain specific language [29] has been designed to write SCR specifications using the four variable model, as well as a large number of tools have been developed to help in checking consistency of the requirement specifications [23, 22]. While the consistency of requirement specifications can be checked using these tools, a hurdle still remains in having absolute confidence in the final system obtained. This hurdle pertains to ensuring that the compilation process is provably correct, i.e., after consistency checking, when the SCR specification is translated into executable code then making sure that the code generated is faithful to the original specification.

5.3 An Interpreter & a Compiler for SCR

We have applied our method discussed in this paper to overcome this hurdle. A Horn logical denotational semantics has been defined for the complete SCR language. This semantics consists of the syntax specification, semantic algebra and valuation predicates. The semantic algebra consists of operations for accessing and updating the store (values of variables as well as their type) and maintaining the various environments.

The grammar of SCR consists of five sections (type definitions, constant definitions, variable declarations, assumptions and assertions, and function definitions). User-defined data types are listed in the type definitions section. There are two types of user define data types: (i) enumerated type and (ii) integer type associated with a range. Variable declarations can include four types of variables: monitored variables, controlled variables, term variables and mode classes. The assumptions and assertions section contains predicates describing relations between variables, i.e., each assumption or assertion is a logical formula. The violation of an assumption indicates that the input does not obey the assumed environmental constraints. If an assertion is violated, it means that the specification does not satisfy a property that is was expected to satisfy. Function in SCR are defined by either a condition or an event table. Functions are used to update values of dependent variables when a monitored variable changes. The DCG for SCR has been developed in accordance to the BNF grammar supplied to us by Naval Research Lab researchers. The semantics of SCR's DSL is given in terms of the store semantic algebra extended with type information.

To illustrate generation of code for SCR in a provably correct manner, we consider a simplified version of the control system for safety injection described in [23, 24]. The system uses three sensors to monitor water pressure and adds coolant to the reactor core when the pressure falls below some threshold. The system operator blocks safety injection by turning on a "block" switch and resets the system after blockage by turning on a "reset" switch. Figure 6 shows how SCR constructs could be used to specify the requirements of the control system. Water pressure and the "block" and "reset" switches are represented as monitored variables, `WaterPres`, `Block`, and `Reset`. Safety injection is represented as a controlled variable, `SafetyInjection`. Each sensor represents an input and the hardware interface between the control system software and the safety injection system serves as output. The program corresponding to this system written in the SCR domain specific language was also supplied to us by researchers at the Naval Research Labs and is shown in Appendix I.

A complete semantics for the SCR domain specific language was developed. Development of the whole semantics required just a few weeks of work (a significant part of this time was spent understanding the SCR language). This Horn logical semantics developed for SCR immediately provides us with an interpreter on which the program above can be executed. Further, the interpreter was partially evaluated w.r.t. this program using the Mixtus system, and compiled code was obtained. The partially evaluated code that corresponds to the safety injection system is shown in Appendix II. The whole partial evaluation (using definite clause semantics of SCR) required 27.1 seconds on a Sun Fire 880 with 150 MHz clock-speed and 1 CPU and 2 GB memory and

Figure 5: The Four-Variable Model

Figure 6: Requirements Spec. for Safety Injection

generated 367 lines of assembly code in Prolog syntax. In [29], given the same example code (Appendix I), a *relation-based* strategy (that associates C code as an attribute with parse tree nodes) required 20 minutes to generate C code, while a transformation-based method using the APTS system [35] took four minutes to generate 293 lines of C code (execution done on a SUN Ultra 450 with 2 UltraSPARC-II 296MHz CPUS and 2GB memory, running Solaris 5.6 [29]).

Even though our respective experiments have been done on different machines, the time taken in our case is considerably better. Note that we did not optimize the semantics at all to make it more amenable to partial evaluation as that would have reduced the readability of the semantics.

5.4 Generating SCR's Debugger

A very simple command language that can be used to examine state during the debug mode has also been designed for SCR. This language consists of four commands:

1. `watch all`: value of each variable in the store is displayed.
2. `watch variable-list`: value of each variable in the list `variable-list` is displayed.
3. `evaluate expression`: `expression` is evaluated and its value output; the expression may contain variables defined in the program being debugged.
4. `exit`: control is returned to the interpreter.

Consider the SCR specification shown in Appendix I. There are two break points in that specification: the first one breaks execution without any condition, the second one breaks when the condition is satisfied. The top level valuation predicate is `program` which takes an input store and produces an output store. It should be noted that developing the interpreter for the command language for debugging and integrating into the semantics of SCR required only a few man-hours of work.

```
?-program([(mcPressure,'TooLow'),
   (mWaterPres,899),
           (prime_mWaterPres,906),
   (prime_mReset,'Off'),
           (mReset,'On'),(mBlock,'Off'),
   (prime_mBlock,'On')],NS).
```

```
Entering debug mode:
Type "help." to get command help.
Type "exit." to quit debugging.
Please input your command: help.

Debug Command Help:
  [exit.]: quit debugger
  [watch all.]: output value of all variables
  [watch variables.]: output values of variables
  [evaluate expression.]: compute an expression
  [help.]:  get system help

Please input your command:watch all.
Low = 900
Permit = 1000
mWaterPres = 899
prime_mWaterPres = 906
mReset = On
prime_mReset = Off
mBlock = Off
prime_mBlock = On
cSafety_Injection = On
prime_cSafety_Injection = On
tOverridden = false
prime_tOverridden = false
mcPressure = TooLow
prime_mcPressure = TooLow
A1 = true

Please input your command:watch mBlock.
mBlock = Off

Please input your command:evaluate (mBlock=On).
Answer : false

Please input your command:exit.
exit debug
```

6 Related Work

Most research work on developing provably correct compilers falls into three classes: (i) those that treat the compiler as a program and verify its correctness manually or semi-automatically using traditional verification techniques, (2) those that generate a compiler automatically from its semantic definition, and (3) those that operationally specify a compiler via rewrite rules.

Considerable work has been done in manually or semi-mechanically proving compilers correct. Most of these efforts are based on taking a specific compiler and showing its implementation to be correct. A number of tools (e.g., a theorem prover) may be used to semi-mechanize the proof. Example of such efforts range from McCarthy's work in 1967 [31] to more recent ones [14, 13, 37, 21, 10]. As mentioned earlier, these approaches are either manual or semi-mechanical, requiring human intervention, and therefore not completely reliable enough for engineering high-assurance systems.

Considerable work has also been done on generating compilers automatically from language semantics. [33, 44, 9, 34, 28, 7]. However, because the syntax is specified as a (non-executable) BNF and semantics is specified using λ-calculus, the automatic generation process is very cumbersome. The approach outlined in this paper falls in this class, except that it uses Horn logical semantics which, we believe and experience suggests, can be manipulated more efficiently.

Considerable work has also been done in using term rewriting systems for transforming source code to target code. In fact, this approach has been applied by researchers at NRL to automatically generate C code from SCR specification using the APTS [35] program transformation system. As noted earlier, the time taken is considerably more than in our approach. Other approaches that fall in this category include the HATS system [45] that use tree rewriting to accomplish transformations. Other transformation based approaches are mentioned in [29].

An approach that comes the close to our approach is Stepney's who also bases her work on logic programming [38, 39]. In Stepney's approach the (provably correct) compiler is specified by the user and not automatically generated as in our framework. The compiler is specified by giving the semantics of program constructs in terms of machine instructions (i.e., the valuation functions map parse trees to machine instructions). The semantics when executed for a given program yields the target code. Consel has independently applied denotational semantics and partial evaluation to the specification and implementation of domain specific languages [8], but his work is based on the traditional model of denotational semantics employing the λ-calculus.

7 Conclusions

In this paper we presented an approach based on denotational semantics, Horn logic, and partial evaluation for obtaining provably correct compiled code for programs. We illustrated our approach in the context of the SCR method for specifying real-time embedded system. A complete syntax and semantic specification for SCR was developed and used for automatically generating code for SCR specifications. Our method produces executable code considerably faster than other transformation based methods for automatically generating code for SCR spec-

ifications. A debugger for SCR was also automatically obtained.

Acknowledgments

We are grateful to Constance Heitmeyer and Elizabeth Leonard of the Naval Research Labs for providing us with the BNF grammar of SCR and the safety injection program as well as for discussions.

References

[1] J. Bentley. Little Languages. *CACM*, 29(8):711-721, 1986.

[2] P. Hudak. Modular Domain Specific Languages and Tools. In *IEEE Software Reuse Conf.* 2000.

[3] C. Ramming. *Proc. Usenix Conf. on Domain-Specific Languages.* Usenix, 1997.

[4] N. G. Leveson, M. P. E. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Software Engineering - ESEC/FSE*, Springer Verlag, pages 127-145, 1999.

[5] W. Codenie, K. De Hondt, P. Steyaert and A. Vercammen. From custom applications to domain-specific frameworks. In *Communications of the ACM*,Vol. 40, No. 10, pages 70-77, 1997.

[6] T. Alspaugh et al. Software requirements for the A-7E aircraft. Tech. Rep. NRL-9194, Naval Research Lab., Wash., DC, 1992.

[7] D.F. Brown et al. ACTRESS: an action semantics directed compiler generator. In *Proc. 4th Int'l Conf. on Compiler Construction*. Springer LNCS 641, pp. 95-109. 1992.

[8] C. Consel. Architecturing Software Using a Methodology for Language Development. In *Proc. 10th Int'l Symp. on Prog. Lang. Impl., Logics and Programs*, '98, Springer LNCS 1490, pp. 170-194.

[9] Thierry Despeyroux. Executable specification of static semantics. In *Semantics of Data Types*, Springer LNCS 173. pp. 215-234. 1984.

[10] A. Dold, T. Gaul, W. Zimmermann Mechanized Verification of Compiler Backends Proc. of the Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark, July 12-13, 1998.

[11] S. R. Faulk. State Determination in Hard-Embedded Systems. Ph.D. Thesis, Univ. of NC, Chapel Hill, NC, 1989.

[12] Y. Futamura. Partial Evaluation of Computer Programs: An approach to compiler-compiler. *J. Inst. Electronics and Communication Engineers, Japan.* 1971.

[13] D. Gladstein and M. Wand. Compiler Correctness for Concurrent Languages. In *Proc. Coordination '96*. Springer LNCS 1061. pp. 231–248, 1996.

[14] Andrew D. Gordon, et al. Compilation and Equivalence of Imperative Objects", Proc. FSTTCS'97, Springer Verlag pp. 74-87.

[15] G. Gupta Horn Logic Denotations and Their Applications, *The Logic Programming Paradigm: A 25 year perspective*. Springer LNAI. 1999:127-160.

[16] Q. Wang, G. Gupta. Horn Logic Continuation Semantics. 2003 Conference on Logic Program Synthesis and Transformation (LOPSTR). Springer Verlag.

[17] G. Gupta, E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Systems Symposium*, pp. 230-239. Dec. 1997.

[18] G. Gupta, E. Pontelli, R. Felix-Cardenas, A. Lara, Automatic Derivation of a Provably Correct Parallelizing Compiler, *In Proceedings of International Conference on Parallel Processing*, IEEE Press, Aug, 1998, pp. 579-586.

[19] G. Gupta, E. Pontelli. A Logic Programming Framework for Specification and Implementation of Domain Specific Languages. In *Essays in Honor of Robert Kowalski*, 2003, Springer Verlag, to appear.

[20] C. Gunter. Programming Language Semantics. MIT Press. 1992.

[21] J. Hannan, F. Pfenning. Compiler Verification in LF. *Proc. Seventh Annual IEEE Symposium on Logic in Computer Science*. pp. 407–418. 1992.

[22] Constance L. Heitmeyer,A. Bull, C. Gasarch, and Bruce G. Labaw. SCR*: A toolset for specifying and analyzing requirements. Proc. 10th Computer Assurance Conf.(COMPASS'95), Gaithersburg, MD, June 1995.

[23] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. ACM Trans. Software Eng. and Methodology 5, 3, July 1996.

[24] C. Heitmeyer, J. Kirby, Jr., et al. Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Transactions on Software Engineering, 24(11):927-948, November 1998.

[25] K. L. Henninger. Specifying software requirements for complex systems: New techniques and their application. IEEE Trans. on Software Engineering. SE-5, 1. pp. 2-13.

[26] L. King, G. Gupta, E. Pontelli. Verification of BART Controller: An Approach based on Horn Logic and Denotational Semantics. In *High Integrity Software*, V. Winter and S. Bhattacharya (eds), April 2001, Kluwer Academic.

[27] N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503.

[28] P. Lee. Realistic Compiler Generation. The MIT Press, Cambridge, MA, 1989.

[29] E. I. Leonard and C. L. Heitmeyer. Program Synthesis from Requirements Specifications Using APTS. Kluwer Academic Publishers, 2002.

[30] K. Marriott and P. Stuckey. Constraint Programming. MIT Press, 1998.

[31] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. MIT AI Lab Memo, 1967.

[32] S. P. Miller. Specifying the model Logic of a Flight Guidance Systems in CoRE and SCR. Pages:44-53 Series Proceeding Article, 1998

[33] P.D. Mosses. Compiler Generation using Denotational Semantics. In *Math. Foundations of CS*, Springer LNCS 45, pages 436-441, 1976.

[34] F. Nielson and H. R. Nielson. Two level semantics and code generation. *Theoretical Computer Science*, 56(1):59-133. 1988.

[35] R. Paige. Viewing a Program Transformation System at Work. *Proc. Programming Language Implementation and Logic Programming*, Springer, LNCS 844. 1994.

[36] E. Pontelli, D. Ranjan, G. Gupta, B. Milligan. ΦLog: A Language for Programming Phylogenetic Inference Problems. Proc. First IEEE Computer Society Bioinformatics Int'l Conf., 2002. pp. 1-10.

[37] C. Pusch. Verification of Compiler Correctness for the WAM. *Proc 9th International Conference on Theorem Proving in Higher Order Logics (TPHOL'96)*. Springer-Verlag LNCS 1125, 1996.

[38] S. Stepney. High Integrity Compilation. Prentice Hall. 1993.

[39] S. Stepney. Incremental Development of a High Integrity Compiler: experience from an industrial development. In Third IEEE High-Assurance Systems Engg. Symp. (HASE'98).

[40] D. Sahlin. An Automatic Partial Evaluator for Full Prolog. Ph.D. Thesis. 1994. Royal Institute of Tech., Sweden. (available at www.sics.se)

[41] D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Publishers, 1986.

[42] A. J. van Schouwen, D. Parnas, J. Madey. Documentation and requirements for computer systems. In *Proc. of Int'l Symp. on Requirements Engineering*, IEEE Computer Society Press. 1993.

[43] L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.

[44] M. Wand. Semantics-directed Machine Architecture. In *ACM POPL*. pp. 234-241. 1982

[45] V. L. Winter. Program Transformation in HATS. *Software Transformation Systems Workshop*, '99.

[46] V. Winter et al. Bay Area Rapid Transit District Advance Automated Train Control System: Case Study Description. In *High Integrity Software*, V. Winter and S. Bhattacharya (eds), April 2001, Kluwer Academic.

[47] V. Wiels and S. Easterbrook. Formal Modeling of Space Shuttle Software Change Requests using SCR. Proceedings of the Fourth International Symposium on Requirements Engineering, Limerick, Ireland, June 7-11,1999.

Appendix I: Example SCR Code

```
spec Safety_Injection_System
type definitions
  ySwitch: enum in {Off, On};
  type_mcPressure: enum in {TooLow, Permitted,
High};
  yWPres: integer in [0, 2000];
constant definitions
  Low=900:integer;
  Permit=1000:integer;
monitored variables
  mWaterPres: yWPres, initially 0;
  mBlock, mReset: ySwitch, initially Off;
controlled variables
  cSafety_Injection: ySwitch, initially On;
term variables
  tOverridden: boolean, initially false;
mode classes
  mcPressure:  type_mcPressure, initially TooLow;
assumptions
   A1: (mWaterPres' >= mWaterPres AND mWaterPres'
-  mWaterPres <=10) OR
       (mWaterPres' < mWaterPres AND mWaterPres -
mWaterPres' <= 10)
function definitions
break;
var mcPressure :=
  case mcPressure
    [] TooLow
      ev
       [] @T(mWaterPres >= Low) -> Permitted
      ve
    [] Permitted
      ev
       [] @T(mWaterPres >= Permit) -> High
       [] @T(mWaterPres <  Low) -> TooLow
      ve
    [] High
      ev
       [] @T(mWaterPres < Permit) -> Permitted
      ve
  esac
break(mBlock=On);
var tOverridden  :=
   ev
    [] @T(mBlock=On) WHEN mReset=Off
          AND NOT ( mcPressure = High) -> true
    [] @T(mReset=On) WHEN NOT ( mcPressure = High)
          OR @T(mcPressure = High)
          OR @T(NOT (mcPressure = High) ) -> false
   ve
var cSafety_Injection ==
   case mcPressure
     [] TooLow
       if
         [] tOverridden -> Off
         [] NOT tOverridden -> On
       fi
     [] Permitted, High
       if
        []  true -> Off
        []  false -> On
       fi
      esac
```

Appendix II: 'Generated Code'

Note: Appendix II is included for reviewers' convenience and will be removed from the final paper.

```
program(A, B) :-
        program1(A, B).
program1(A, _) :-
        initialize_store,
        set('Low', 900),
        set(prime_Low, 900),
        set('Permit', 1000),
        set(prime_Permit, 1000),
        set(mWaterPres, 0),
        set(prime_mWaterPres, 0),
        set(mReset, 'Off'),
        set(prime_mReset, 'Off'),
        set(mBlock, 'Off'),
        set(prime_mBlock, 'Off'),
        set(cSafety_Injection, 'On'),
        set(prime_cSafety_Injection, 'On'),
        set(tOverridden, false),
        set(prime_tOverridden, false),
        set(mcPressure, 'TooLow'),
        set(prime_mcPressure, 'TooLow'),
        readInputVar(A),
        access(prime_mWaterPres, B),
        access(mWaterPres, C),
        (   C=<B ->
            D=true
        ;   D=false
        ),
        access(prime_mWaterPres, E),
        access(mWaterPres, F),
        G is E-F,
        (   10<G ->
            H=false
        ;   H=true
        ),
        (   D==true,
            H==true ->
            I=true
        ;   I=false
        ),
        access(prime_mWaterPres, J),
        access(mWaterPres, K),
        (   J<K ->
            L=true
        ;   L=false
        ),
        access(mWaterPres, M),
        access(prime_mWaterPres, N),
        O is M-N,
        (   10<O ->
            P=false
        ;   P=true
        ),
        (   L==true,
            P==true ->
            Q=true
        ;   Q=false
        ),
        (   I==false,
            Q==false ->
            R=false
```

```
;   R=true
),
set('A1', R),
access(mcPressure, S),
(   S='TooLow' ->
    access(prime_mWaterPres, T),
    access(prime_Low, U),
    (   U=<T ->
        V=true
    ;   V=false
    ),
    access(mWaterPres, W),
    access('Low', X),
    (   X=<W ->
        Y=true
    ;   Y=false
    ),
    (   Y==false,
        V==true ->
        Z=true
    ;   Z=false
    ),
    (   Z==true ->
        A1='Permitted'
    ;   A1=none
    )
;   A1=none
),
(   A1==none ->
    (   S='Permitted' ->
        access(prime_mWaterPres, B1),
        access(prime_Permit, C1),
        (   C1=<B1 ->
            D1=true
        ;   D1=false
        ),
        access(mWaterPres, E1),
        access('Permit', F1),
        (   F1=<E1 ->
            G1=true
        ;   G1=false
        ),
        (   G1==false,
            D1==true ->
            H1=true
        ;   H1=false
        ),
        (   H1==true ->
            I1='High'
        ;   I1=none
        ),
        (   I1==none ->
            access(prime_mWaterPres, J1),
            access(prime_Low, K1),
            (   J1<K1 ->
                L1=true
            ;   L1=false
            ),
            access(mWaterPres, M1),
            access('Low', N1),
            (   M1<N1 ->
                O1=true
            ;   O1=false
            ),
            (   O1==false,
                L1==true ->
                P1=true
            ;   P1=false
            ),
            (   P1==true ->
                Q1='TooLow'
            ;   Q1=none
            )
        ;   Q1=I1
        )
    ;   Q1=none
    ),
    (   Q1==none ->
        (   S='High' ->
            access(prime_mWaterPres, R1),
            access(prime_Permit, S1),
            (   R1<S1 ->
                T1=true
            ;   T1=false
            ),
            access(mWaterPres, U1),
            access('Permit', V1),
            (   U1<V1 ->
                W1=true
            ;   W1=false
            ),
            (   W1==false,
                T1==true ->
                X1=true
            ;   X1=false
            ),
            (   X1==true ->
                Y1='Permitted'
            ;   Y1=none
            )
        ;   Y1=none
        )
    ;   Y1=Q1
    )
;   Y1=A1
),
(   Y1==none ->
    access(mcPressure, Z1),
    update(prime_mcPressure, Z1)
;   update(prime_mcPressure, Y1)
),
access(prime_mBlock, A2),
(   A2=='On' ->
    B2=true
;   B2=false
),
access(mBlock, C2),
(   C2=='On' ->
    D2=true
;   D2=false
),
(   D2==false,
    B2==true ->
    E2=true
;   E2=false
),
access(prime_mReset, F2),
(   F2=='Off' ->
    G2=true
;   G2=false
```

```
),
access(prime_mcPressure, H2),
(   H2=='High' ->
    I2=true
;   I2=false
),
(   I2==true ->
    J2=false
;   J2=true
),
(   G2==true,
    J2==true ->
    K2=true
;   K2=false
),
(   E2==true,
    K2==true ->
    L2=true
;   L2=false
),
(   L2==true ->
    M2=true
;   M2=none
),
(   M2==none ->
    access(prime_mReset, N2),
    (   N2=='On' ->
        O2=true
    ;   O2=false
    ),
    access(mReset, P2),
    (   P2=='On' ->
        Q2=true
    ;   Q2=false
    ),
    (   Q2==false,
        O2==true ->
        R2=true
    ;   R2=false
    ),
    access(prime_mcPressure, S2),
    (   S2=='High' ->
        T2=true
    ;   T2=false
    ),
    (   T2==true ->
        U2=false
    ;   U2=true
    ),
    (   R2==true,
        U2==true ->
        V2=true
    ;   V2=false
    ),
    access(prime_mcPressure, W2),
    (   W2=='High' ->
        X2=true
    ;   X2=false
    ),
    access(mcPressure, Y2),
    (   Y2=='High' ->
        Z2=true
    ;   Z2=false
    ),
    (   Z2==false,
        X2==true ->
        A3=true
    ;   A3=false
    ),
    access(prime_mcPressure, B3),
    (   B3=='High' ->
        C3=true
    ;   C3=false
    ),
    (   C3==true ->
        D3=false
    ;   D3=true
    ),
    access(mcPressure, E3),
    (   E3=='High' ->
        F3=true
    ;   F3=false
    ),
    (   F3==true ->
        G3=false
    ;   G3=true
    ),
    (   G3==false,
        D3==true ->
        H3=true
    ;   H3=false
    ),
    (   A3==false,
        H3==false ->
        I3=false
    ;   I3=true
    ),
    (   V2==false,
        I3==false ->
        J3=false
    ;   J3=true
    ),
    (   J3==true ->
        K3=false
    ;   K3=none
    )
;   K3=M2
),
(   K3==none ->
    access(tOverridden, L3),
    update(prime_tOverridden, L3)
;   update(prime_tOverridden, K3)
),
access(mcPressure, M3),
(   M3='TooLow' ->
    access(prime_tOverridden, N3),
    (   N3==true ->
        O3='Off'
    ;   O3=none
    ),
    (   O3==none ->
        access(prime_tOverridden, P3),
        (   P3==true ->
            Q3=false
        ;   Q3=true
        ),
        (   Q3==true ->
            R3='On'
        ;   R3=none
        )
```

```
    ;   R3=O3
    )
;   R3=none
),
(   R3==none ->
    (   M3='High' ->
        S3='Off'
    ;   M3='Permitted' ->
        S3='Off'
    ;   S3=none
    )
;   S3=R3
),
(   S3==none ->
    access(cSafety_Injection, T3),
    update(prime_cSafety_Injection, T3)
;   update(prime_cSafety_Injection, S3)
).
```