Higher-Order and Symbolic Computation, 16, 63–92, 2003

Program Synthesis from Formal Requirements Specifications Using APTS*

ELIZABETH I. LEONARD leonard@itd.nrl.navy.mil
CONSTANCE L. HEITMEYER heitmeyer@itd.nrl.navy.mil
Naval Research Laboratory, Code 5546, Washington, DC 20375, USA

Abstract. Formal specifications of software systems are extremely useful because they can be rigorously analyzed, verified, and validated, giving high confidence that the specification captures the desired behavior. To transfer this confidence to the actual source code implementation, a formal link is needed between the specification and the implementation. Generating the implementation directly from the specification provides one such link. A program transformation system such as Paige's APTS can be useful in developing a source code generator. This paper describes a case study in which APTS was used to produce code generators that construct C source code from a requirements specification in the SCR (Software Cost Reduction) tabular notation. In the study, two different code generation strategies were explored. The first strategy uses rewrite rules to transform the parse tree of an SCR specification into a parse tree for the corresponding C code. The second strategy associates a relation with each node of the specification parse tree. Each member of this relation acts as an attribute, holding the C code corresponding to the tree at the associated node; the root of the tree has the entire C program as its member of the relation. This paper describes the two code generators supported by APTS, how each was used to synthesize code for two example SCR requirements specifications, and what was learned about APTS from these implementations.

Keywords: code generation, code synthesis, APTS, SCR, requirements specifications, formal specifications, program transformation

1. Introduction

In developing complex software, an operational specification of the system's required behavior can be extremely useful. Such a specification can be (1) formally verified to show that critical properties are satisfied and (2) validated using simulation to show that the intended system behavior is captured. Additionally, because specifications contain much less detail than programs, errors are easier to find in specifications than in programs, and specifications are easier to understand than programming language code. Thus, we can develop confidence that a specification is correct. Unfortunately, high assurance in the correctness of the specification does not mean that the implementation of the system is correct, since the implementation is usually separately developed with no formal link to the specification. Some confidence in the correctness of the actual code can be achieved by testing, but that confidence is only as good as the tests used. One way to transfer high confidence in the specification to the implementation is to automatically generate the implementation code from the specification, thus eliminating the errors introduced by hand-coding.

*This research was funded by the Office of Naval Research.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **2003**	2. REPORT TYPE	3. DATES COVERED **00-00-2003 to 00-00-2003**

4. TITLE AND SUBTITLE **Program Synthesis from Formal Requirements Specifications Using APTS***	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Code 5546,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**30**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

A program transformation system such as Cai and Paige's APTS [7, 33, 34] can be extremely useful in developing an automatic code generator. APTS, which is implemented in SETL2 [40], is an improved version of Paige's earlier RAPTS system [31]. APTS includes a syntax analyzer, a relational database, and a transformation engine. The syntax analyzer uses a given grammar to parse a specification. The relational database accumulates information about the specification needed during code generation. Code can be generated either in the relational database or via the transformation engine. APTS also includes optimization techniques, such as *finite differencing* [31, 32], a technique for optimizing code by replacing frequently repeated calculations with less expensive incremental updates.

Cai and Paige used APTS [7] to implement translators from SETL2 [40] and SQ2+ [6] to C. This paper describes a case study in which APTS was used to generate C code from requirements specifications in the SCR (Software Cost Reduction) tabular notation. Using APTS, two code generation strategies were implemented. One strategy used only the relational database, and the other used both the relational database and the transformation engine.

SCR is a formal method for specifying the required behavior of software systems. The SCR toolset provides a user-friendly interface for writing requirements specifications in a tabular format and a number of analysis tools, including a consistency checker [18], a simulator [17], a model checker [16], a theorem prover [2], and an invariant generator [21, 24]. In the toolset, the specification is displayed as a collection of tables. A context-free grammar is the underlying communication medium for the different tools. By applying the SCR tools, a user can develop high confidence that a specification is a correct statement of the required system behavior.

The SCR method has been used successfully by many organizations in industry and in government (e.g., Bell Laboratories [20], Grumman [29], Lockheed [10], the Naval Research Laboratory [16, 25], Ontario Hydro [35], and Rockwell Aviation [30]) to develop and analyze specifications of practical systems, including flight control systems [10, 30], weapons systems [16], space systems [9], and cryptographic devices [25]. Most recently, the SCR tools were used by Lockheed Martin, together with a test case generator, to detect a critical error described as the "most likely cause" of a $165M failure in the software controlling landing procedures in the Mars Polar Lander [5].

The next step in developing high assurance systems using SCR is to synthesize (i.e., generate) code from the SCR specifications. This paper describes an initial step toward this goal that uses APTS to explore two possible strategies for code generation from SCR requirements specifications. The first strategy uses rewrite rules to transform code in the source language into target language code. The source language used for these experiments was the SCR specification language and the target language was C. The transformations modify the parse tree of the requirements specification, replacing each node containing SCR code with a new node containing the corresponding C code. The second strategy treats the code to be generated as a synthesized attribute of the parse tree of the SCR specification. A relation is developed that associates target language code with each node in the parse tree of the source language program. In the relational model, the code associated with a given node is formed by combining the code for the node's children with additional code specific for the node and the result is stored in a relation. In both strategies, auxiliary information (for example, variable dependencies) is stored in relations. Frequently a node of the parse tree is the key for the relation. In these cases, the relations are analogous to the attributes in an attribute grammar [26]. Both code synthesizers were implemented using the three components of APTS described above.

The use in APTS of a relational database to store attribute information has advantages over traditional attribute grammars. In APTS, information can be passed directly from one node to any other node via pattern matching in the relational database, while in an attribute grammar, information must flow along a path through the tree. Although any attribute grammar can be augmented with auxiliary data structures to hold inherited attributes for lookup, the relational database used by APTS is an integral part of the system, holding all attribute information. Also, APTS allows relations to be defined over domains other than nodes of the parse tree. For example, the tree domain allows attribute information to be assigned to all nodes representing the same term. Such a tree relation can be used to assign the same attribute value to all occurrences of a variable at the same time. In contrast, in a traditional attribute grammar, this information must be passed around the tree.

Our study used APTS because its flexible framework allowed us to experiment with two different code generation strategies. Also, because both approaches required much of the same information about an SCR specification, we were able to reuse a large portion of the first generator in the implementation of the second one. Similar experiments could have been conducted with other systems, such as REFINE [37] or the Synthesizer Generator [38], but both rely on attribute grammars and thus suffer from the restrictions on information flow and absence of built-in auxiliary data structures described above. An additional advantage of APTS over the Synthesizer Generator is that program transformations in APTS are automatic rather than interactive as in the Synthesizer Generator and may be conditioned on information stored in the relations. However, the primary advantage of APTS over other systems is its built-in optimization capabilities. Although APTS' finite differencing capabilities are not used in our current implementations, in the future we plan to apply optimizations to the specifications by augmenting our APTS-based code generators to use the APTS finite differencing engine.

This paper is organized as follows. Section 2 reviews APTS and SCR and describes the C code that can be generated from SCR specifications. Section 3 describes in more detail the two strategies for generating C code described above, one strategy based on rewrite rules and the second based on the accumulating relation approach. Section 4 describes the results of applying the code generators to an SCR requirements specification of a cryptographic device [25]. It also compares the two strategies and discusses what we learned by implementing them in APTS. Section 5 discusses related work. Finally, Section 6 presents some conclusions and describes our plans for future work.

2. Background

2.1. APTS

From a given grammar, the APTS syntax analyzer builds parse trees for input files. Each node of the parse tree forms the root of a subtree corresponding to the SCR syntax for that portion of the specification. The grammar used to specify SCR in APTS is similar to the abstract grammar described in Appendix A, although in a few places the form of the grammar used in APTS was modified to allow more convenient transformational processing. For example, the variable declarations in the abstract grammar are defined by the grammar rule *var_decls* ::= *id* (, *id*)* : *type*, **initially** *init_val*. Although this syntax is allowed in APTS, our implementations use the following pair of grammar rules.

```
scr_var_decl = declist ':' type ',' 'initially'
               init_val ';' ;
declist = id  | id ',' declist;
```

This change was made because some of the APTS relational database and rewrite rules need to refer to each identifier in the list individually. In such cases, this alternate form of the grammar is easier to use.

The relational database is built using a set of inference rules. These rules usually consist of three parts: a pattern to match a portion of the parse tree, a set of conditions (logical combinations of relations) under which a new member of a relation is added to the database, and a result portion stating which new member to add to a relation in the database. Pattern matching variables, whose names in APTS begin with a dot, are used to correlate portions of the pattern matching condition with the relations appearing in the rule's condition and result portions or to correlate information contained in different relations in the rule. For example, the following rule states that if an expression composed of two subexpressions separated by a ">" is found in the parse tree such that the two subexpressions are members of the **arithexpr** relation, then the matched expression (denoted by `loc()`) is added to the relation **boolexpr**:

```
match(%expr, .x > .y%)| arithexpr(.x) and arithexpr(.y)
  -> boolexpr(loc());
```

Relations are predicates whose arguments can be taken from several possible domains. Frequently, nodes in the parse tree serve as keys for relations. Thus, these relations may be viewed as attributes of the parse tree. Since there are no restrictions on which relations can be in the condition of an inference rule for a relation, both synthesized and inherited attributes can be defined in APTS in terms of relations. However, not every relation in the implementation need be a parse tree attribute. For example, in SCR specifications, the value of a variable may depend on the current value of another variable. During execution of the corresponding C code, the values of variables must be updated in an order that respects these dependencies. Thus, this dependency information is necessary for code generation. In the implementations, the dependency relation between variables is implemented as a relation defined over pairs of strings (the variable names). Relations in APTS are grouped together into *transcripts*. To execute a transcript, the inference engine partially instantiates all applicable rules from the transcript and then nondeterministically tries to complete them until there are no more rules that can be instantiated.

APTS also allows user-defined SETL2 routines to be used to add relations to the database. The user specifies an interface in APTS for each SETL2 routine, stating which relations the routine receives as input and the relations it will produce upon execution. The SCR-to-C code generators use a SETL2 routine to calculate the order in which variables are to be updated based on the information in the dependency relation.

In addition to using the relational database, the code generator based on rewrite rules also uses the APTS transformation engine. It uses a set of rewrite rules to transform the parse tree, replacing SCR language constructs with the corresponding C code. These rewrite rules match a pattern in the tree and if the conditions for the rule are met, the tree is transformed into the given replacement tree. For example, the rule `equal` below states that if an expression consisting of two subexpressions separated by "=" is found in the parse tree, then that piece of the tree is rewritten in C-style, with the "=" replaced by "==". Rewrite rules can be read as matching a piece of the parse tree and if the given conditions are met, the matched part of the parse tree is replaced by the tree given inside the rewrite.

```
equal: match(%expr, .x = .y%) |true ->
  rewrite(%expr, .x == .y%);
```

In APTS, rewrite rules are collected into groups called *closures*. When a closure is applied to the parse tree, the transformation engine works bottom up on the tree, nondeterministically applying rules from the closure to the tree until no more rules can be applied. In our implementation, we apply closures in a certain order to guarantee that certain rules will be applied before others.

2.2. *SCR specifications*

Originally formulated to document the requirements of the flight program of the U.S. Navy's A-7 aircraft [19], the SCR requirements method is designed to detect and correct errors during the requirements phase of software development [15, 18]. In SCR, the required behavior of a software system is defined in terms of *monitored* and *controlled variables*, which represent quantities in the system environment that the system monitors and controls. A set of *assumptions* describes the constraints imposed on the monitored and controlled quantities by physical laws and the system environment, and a relation on the monitored and controlled variables describes how the system is required to change the values of the controlled quantities in response to changes in the values of the monitored quantities. A set of *assertions* describes properties, such as security and safety properties, that the specification is expected to satisfy.

To specify the required behavior of a software system in a practical and efficient manner, the A-7 requirements document introduced two kinds of predicates—conditions and events—and two kinds of auxiliary variables—mode classes and terms. Conditions and events are defined in terms of the system state, where a *system state* is a function that maps each *state variable* (a monitored, controlled, or auxiliary variable) to a type-correct value. A *condition* is a predicate defined on a single system state, while an *event* is a predicate defined on two system states that denotes some change in the values of the state variables between those states. An event "occurs" if it evaluates to true for a given pair of consecutive states. A *monitored event* occurs when the value of a monitored variable changes. A *conditioned* *event*, which has the form "`@T(`c`) WHEN` d," occurs if an event occurs (i.e., condition c becomes true) when a specified condition d is true. A *mode class* may be viewed as a state machine, whose states are called *modes* and whose transitions are triggered by events. A *term* is a state variable, defined in terms of monitored variables, mode classes, or other terms. Mode classes and terms capture history—the changes that occurred in the values of the monitored variables—and help make the specification more concise.

SCR specifications include two kinds of tables: condition tables and event tables. Each defines the value of a dependent variable (a controlled or auxiliary variable) by means of a mathematical function. Usually, a condition table defines a variable as a function of a mode and a *condition*, and an event table defines a variable as a function of a mode and an *event*.

The purpose of the SCR requirements model [18] is to provide a precise semantics for the notation used in SCR requirements specifications. The model defines a conditioned event "@T(c) `WHEN` d" as

$$@\mathrm{T}(c)\ \texttt{WHEN}\ d \;=\; \neg c \,\wedge\, c' \,\wedge\, d, \tag{1}$$

where c and d are conditions, and the unprimed c denotes c in the old state and the primed c denotes c in the new state. The model also defines the functions that can be derived from the SCR tables. In the SCR model, a software system Σ is represented as a state machine $\Sigma = (S, S_0, E^m, T)$, where S is a set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of monitored events, and T is the transform describing the allowed state transitions. To compute the new state, the transform T composes the functions derived from the condition and event tables. For T to be well-defined, no circular dependencies are allowed in the definitions of the new state variable values. To achieve this, the model requires the new state dependencies (i.e., dependencies among the new values of the state variables) to be a partial order of the state variables.

While an SCR specification is represented as a collection of tables, the underlying communication medium between the SCR tools is a context-free grammar. Appendix A contains an abstract grammar for SCR. This grammar focuses on the basic syntax of the constructs, omitting the precedence rules needed for unambiguous parsing. The abstract grammar is similar to the grammars used by our APTS implementations to parse SCR specifications. However, those grammars contain modifications, including precedence rules, necessary for unambiguous parsing.

The syntax of the language is best illustrated by an example. Below is the specification of a simplified version of a control system for safety injection (SIS) in a nuclear power plant [8]. (The line numbers are not part of the actual specification. They are included for ease of reference.) The SIS system monitors water pressure and if the pressure is too low, the system injects coolant into the reactor core. There are three monitored variables in this specification (lines 9–11).[1] The first is `mWaterPres`, which represents the actual value of water pressure. The other two monitored variables are switches—`mBlock`, a switch that overrides safety injection, and `mReset`, a switch that resets the system after blockage. An assumption of the specification is that the water pressure will not change by more than 10 units at a time (lines 18–21). A mode class `mcPressure`, with three possible values `TooLow`, `Permitted`, and `High`, associates the pressure with the appropriate range (lines 16–17). At any given time, the system must be in one and only one of these modes. The term variable `tOverridden` (lines 14–15) is *true* if safety injection is blocked, and *false* otherwise. The specification contains one controlled variable, `cSafety_Injection`, which represents a switch indicating whether safety injection is on or off (lines 12–13). The value of each dependent variable is defined by a table function. Event tables define the value of the mode class `mcPressure` (lines 24–39) and the term variable `tOverridden` (lines 40–47). A condition table defines the value of the controlled variable `cSafety_Injection` (lines 48–60). The tabular representations of the function definitions on lines 24–60 may be found in Appendix B.

```
 1   spec Safety_Injection_System
 2   type definitions
 3     ySwitch: enum in {Off, On};
 4     type_mcPressure: enum in {TooLow, Permitted,High};
 5     yWPres: integer in [0, 2000];
 6   constant definitions
 7     Low=900:integer;
 8     Permit=1000:integer;
 9   monitored variables
10     mWaterPres: yWPres, initially 0;
11     mBlock, mReset: ySwitch, initially Off;
12   controlled variables
13     cSafety_Injection: ySwitch, initially On;
14   term variables
15     tOverridden: boolean, initially false;
16   mode classes
17     mcPressure:  type_mcPressure, initially TooLow;
18   assumptions
19      A1: (mWaterPres' >= mWaterPres AND mWaterPres'-
20          mWaterPres <=10) OR (mWaterPres' < mWaterPres
21          AND mWaterPres - mWaterPres' <= 10);
22   assertions
23   function definitions
24   var mcPressure :=
25     case mcPressure
26      [] TooLow
27         ev
28            [] @T(mWaterPres >= Low) -> Permitted
29         ve
30      [] Permitted
31         ev
32            [] @T(mWaterPres >= Permit) -> High
33            [] @T(mWaterPres <  Low) -> TooLow
34         ve
35      [] High
36         ev
```

```
         [] @T(mWaterPres < Permit) -> Permitted
      ve
  esac
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
        []  false -> On
        []  true -> Off
      fi
  esac
```

2.3. C code generated from SCR specifications

An SCR specification contains several sections. Code must be generated from each section. Additionally, code is needed to drive the reactive program. In practice, this code would be replaced by the device driver software for the system. This section describes the C code generated from an SCR specification. The format of the C code to be generated is given as part of the grammar file in APTS. Our synthesizers produce code that is very closely related to the SCR specification. This makes correspondence between the specification and the code easy to observe.

Some pieces of code are generated for every specification. At the beginning of each generated code file are two file pointers, `infile` and `outfile`, which will be associated with the input and output files that drive the reactive program. Also included are input and output routines for boolean values (represented internally by the integer constants false = 0 and true = 1).

For every SCR specification, the corresponding C code contains a header file "scr-header.h". This header file is exactly the same for every specification and is not generated by APTS. This file contains definitions of formats for reading and writing strings and integers. For example, the string output format macro is defined as `# define strformout "%s\n"`. Defining these format routines in a header file that is not generated by APTS is necessary because APTS treats the '%' as a special character that cannot appear in relations or rewrite rules. The file also contains definitions used by the generated C code. For example, `boolean` and `integer` are defined as additional names for the C type `int`; and `false, true, AND, OR`, and `NOT` are defined as corresponding C code values and operators. None of these definitions is strictly necessary, but they are included to make the C code resemble the SCR specification.

Type definitions. Both SCR and C support enumerated types. However, unlike C, SCR allows overloading of value names in enumerated types. To handle this soundly in our encoding, we simply prepend the type name to each enumerated value. For example, the value `Off` of the enumerated type `ySwitch` in the SCR specification is transformed into `ySwitch_Off` in the C type definition.

```
/* type definitions and range declarations */
   enum ySwitch { ySwitch_Off , ySwitch_On } ;
   typedef enum ySwitch ySwitch ;
```

Each enumerated type also requires special input and output routines to convert the value names used in the specification to the corresponding value names used in the C code, and vice versa.

The user-defined range types in SCR have no counterpart in C. In the generated C code, the name of the range type becomes an alias for integer, and a check function is created to correspond to the range of the type. Each time a variable with a range type is assigned a value, the corresponding check function is called to ensure that the value is within the specified range. The SIS example contains one range type, `yWPres`, with the range [0,2000]. Below is the C code corresponding to this range type.

```
# define yWPres int
 void check_yWPres (char * name, int value) { if ((value
   < 0) OR (value > 2000)) {printf(" value out of
   range : "); printf (strformout, name); } }
```

Constant definitions. The generated C code for an SCR constant definition is a straightforward rearrangement of the SCR definition. Below is the C code generated for the constant definitions of the SIS example.

```
/* constant definitions */
   const integer Low = 900 ;
   const integer Permit = 1000 ;
```

Variable declarations. In an SCR specification, `x` represents the value of variable x in the old state, and `x'` represents the value of x in the new state. To refer to both the old and new values of the variable x, the generated C code represents each variable `x` in the SCR specification by two variables, `x` and `prime_x`. Initial values, if given,[2] are defined by constants. The name given to these constants is constructed by prepending `init_val_` to the name of the first variable in the declaration. For example, the initial value for `mBlock` and `mReset` is named `init_val_mBlock`. Below is the C code corresponding to the declaration of the monitored variables. The other variable declarations may be transformed into C code in a similar way.

```
/* monitored variables */
  yWPres mWaterPres; yWPres prime_mWaterPres;
     const yWPres init_val_mWaterPres = 0;
  ySwitch mBlock, mReset; ySwitch prime_mBlock,
  prime_mReset;
     const ySwitch init_val_mBlock = ySwitch_Off;
```

Assumptions and assertions. In SCR, assumptions and assertions are predicates describing relationships between the variables. These logical formulas may refer to both the old and new state values of the variables and can use a full range of logical operators. Event expressions may also appear in predicates and are expanded using definition (1). Each assumption or assertion in the specification is transformed into an evaluation function which returns true if the predicate is true and false otherwise. Additionally, two functions, `check_assumptions` and `check_assertions`, which call these functions and produce an error message if a predicate is false, are generated if there are assumptions and assertions in the SCR specification. The violation of an assumption indicates that the input does not obey the assumed environmental constraints. If an assertion is violated, then the specification does not satisfy a property that it was expected to satisfy. In the SIS example below, an evaluation function is generated for the assumption `A1` along with the function `check_assumptions` which calls the evaluation function. There are no assertions in the SIS specification, so the function `check_assertions` is not generated.

```
/* assumptions */
 boolean eval_A1( ) {return((prime_mWaterPres >=
   mWaterPres AND prime_mWaterPres - mWaterPres <= 10 ) OR
   ( prime_mWaterPres < mWaterPres AND mWaterPres -
   prime_mWaterPres <= 10 ) ) ; } ;
 void check_assumptions ( ) {
   if ( eval_A1( ) == false ) printf (" A1 violated \n ");
 }
 /* assertions */
```

Function definitions. As stated in Section 2.2, each dependent variable in an SCR specification is associated with a function. This function is defined by either a condition or an event table, describing how the variable's value is updated when a monitored variable changes. For each SCR table function, the C code contains a corresponding update function in which the successful branches assign the newly calculated value to the primed version of the variable. Each branch of the SCR `case` statement in a condition table becomes a C `if` statement conditioned on the value of the primed version of the mode class variable.

Each SCR `if` statement is transformed into a C `if-else` statement. Below is the C update function corresponding to the condition table for `cSafety_Injection`. Note that no code is generated for the false branch (line 57) in the table for `cSafety_Injection`.[3] The tables for `mcPressure` and `tOverridden` are transformed into C code in a similar way.

```
void update_cSafety_Injection ( ) {
    if (( prime_mcPressure == type_mcPressure_TooLow )) {
        if ( prime_tOverridden ) {
          prime_cSafety_Injection = ySwitch_Off ;
          fprintf ( outfile , " cSafety_Injection = " ) ;
        put_ySwitch ( prime_cSafety_Injection ) ; }
      else if ( NOT prime_tOverridden ) {
        prime_cSafety_Injection = ySwitch_On ;
        fprintf ( outfile , " cSafety_Injection = " ) ;
      put_ySwitch ( prime_cSafety_Injection ) ; }
    } ;
    if ( ( prime_mcPressure == type_mcPressure_Permitted )
      OR ( prime_mcPressure == type_mcPressure_High ) ) {
      if ( true ) {
        prime_cSafety_Injection = ySwitch_Off ;
        fprintf ( outfile , " cSafety_Injection = " ) ;
      put_ySwitch ( prime_cSafety_Injection ) ; }
    } ;
}
```

Execution code. In addition to generating code from the specification, we also generate code which executes the specified state machine. The generated code simulates input and output using text files. Input is from a file which lists monitored events, each specified by the name of a monitored variable and a value to be assigned to that variable. The execution model is similar to the execution model of SCR systems used in the translation of SCR into Promela (the language of the SPIN model checker) by Bharadwaj and Heitmeyer [3] and can be described (in pseudocode) as follows.

```
<open files>
state = 0;
<initialize new state variables>;
<check assumptions and assertions>;
while (<infile contains another monitored event> ) {
    state = state+1;
    <copy new state variables to old state variables>;
    <update new state variable corresponding to monitored event>;
    <update new state dependent variables in dependency order>;
    <check assumptions and assertions>;
}
<close files>
```

Separate functions are generated for performing the initialization, copying the variables, and updating the dependent variables. Note that the dependent variables are updated in an order consistent with the partial order describing the new state dependencies as discussed in Section 2.2. The previously generated check_assumptions and check_assertions functions are also called by this main routine. All SCR specifications generate a similar main routine; the only differences are in the names of the update functions for the dependent variables and the updating of the monitored variables in response to monitored events.

3. Generating C code from SCR specifications

This section describes our implementation of the two strategies for code generation. Both strategies use many of the same relations in their generation of code. Section 3.1 describes these relations. Sections 3.2 and 3.3 describe the strategies, the first using rewrite rules and the second using accumulating relations.

3.1. Relations common to both strategies

To generate C code from an SCR specification, each code generator makes extensive use of the APTS relational database. Relations are defined to compute and store information needed to generate code. In the implementations, some relations have rules that are conditioned on other relations not holding for a node. Thus, relations that appear negated in the conditions of rules need to be fully calculated before the rules that contain those negations can be applied. To accomplish this, the relations are organized into groups called *transcripts* that can be calculated in the same pass. The transcripts are executed in an order that respects the dependencies of relations in one group on relations in another group. Both code generation strategies require similar information to be stored, i.e., the variable dependencies, information about the types of expressions, and several pieces of code that need to be calculated and stored in the nodes before the C code is generated.

Both strategies need information about variable dependencies. The order in which the dependent variables are updated depends on the new state dependencies. This dependency information is calculated by the SCR toolset and then converted into an APTS relation **depend**.[4] In the SIS example, the rules for **depend** are as follows:

```
true -> depend(mcPressure, mWaterPres);
true -> depend(tOverridden, mBlock);
true -> depend(tOverridden, mReset);
true -> depend(tOverridden, mcPressure);
true -> depend(cSafety_Injection, mcPressure);
true -> depend(cSafety_Injection, tOverridden);
```

These rules can be read as mcPressure depends on mWaterPres, tOverridden depends on mBlock, and so on. This relation is passed to a SETL2 routine that constructs a topological sort of the variables with respect to the dependency constraints. The results of this SETL2 routine are stored in a relation **followedby** that holds the ordering of the variables.

Several relations are used to check that the input specification is a valid SCR specification. These relations are necessary because in the parsing grammar, expressions, events, and predicates are condensed into one category. The relations mark the nodes containing each of these separate types of expression. For example, `@T(mBlock = on)` is a member of the **eventexpr** relation, as is `@T(mBlock = on) WHEN (mReset=Off AND NOT (mcPressure=High))`. An additional relation **primeexpr** marks the nodes containing primes. Members of the **primeexpr** relation include `mWaterPres′`, `mWaterPres′ - mWaterPres`, and `mWaterPres′ - mWaterPres <= 10`. Using the information in these relations, checks are done to determine whether expressions including primes, events, and predicates are used only as allowed in the SCR language.

Other relations store information needed to generate the code. For example, each variable in the generated code has a corresponding primed variable. In the relational database, a relation **primename**, associated with a variable in the specification, holds the name to be used for the primed version of the variable.

```
match(%declist, .x,.y%)| true
  -> primename(.x,concat(’prime_’,str(.x)));

match(%declist, .x %) | isavar(rchild(.x))
  -> primename(rchild(.x),concat(’prime_’,str(rchild(.x))));
```

Most APTS relational database rules first match a specific construct in the parse tree, in this case, a list of variables that is part of a declaration. (Recall that, in APTS, pattern matching variables have names beginning with a dot.) In the first rule above for **primename**, if the list has the form of an identifier .x followed by a comma and list of identifiers .y, then we convert the node .x to a string and prepend the string “prime_” to it and associate the resulting string with the tree at .x. (**primename** is a relation between trees and strings, meaning that the string is associated with every instance of the identifier .x in the tree, not just the instance of .x in the node matched by the rule.) The elements in the list .y are assigned their prime names by repeated applications of the rules. The first rule handles all multiple element lists but not single element lists. In the second rule, the match condition states that .x must be a declist. This match condition will match any declist with any number of elements. In the second rule, we only wish to match declists with a single element. The condition that the rightmost child of .x be a variable ensures that the second rule is only triggered by a single element list. (If the list has multiple elements, the right child of .x will be a declist, not a variable.) The actual identifier is the child of node .x and this is what is associated with the new string. Note that when matching more complex patterns, as in the first rule, APTS is able to associate the pattern matching variables with the children of the matched node (e.g., .x is the leftmost child), but when the pattern consists of just a single pattern matching variable, as in the second rule, the pattern matching variable is associated with the matched node rather than with one of its children, even in the case when there is only one child. This makes it necessary to use rchild(.x) to refer to the child in the second rule.

For enumerated type variables, relations hold the names of the relevant input and output routines, as described in Section 2.3. Another relation marks the nodes containing obviously dead code, e.g., branches labeled by `false`, `never`, `@T(true)`, or `@T(false)`. In the SIS example, the branch `[] false -> On` on line 57 is marked as dead code by the rule below. No code is generated for such branches. In the rule, `loc()` refers to the node matched by the pattern matching portion of the rule.

```
match(%if_stmt_body, [] false -> .y%) |true
  -> deadcode(loc());
```

Relations are also used to hold some pieces of the C code. This is done when the code needs to be calculated at one point in the parse tree and generated somewhere else in the tree. The code for these functions is generated during one of the earlier phases of the relational database creation and then passed to the transformation engine or used in the calculation of the code accumulating relation. For example, the code for the C functions that execute the specification must be placed at the end of the generated code. These functions must contain code for each variable in the specification, and thus the code must be calculated in the variable declarations portion of the parse tree because that is where the variables are actually listed. As another example, each value in the list of values for an enumerated type requires code for reading and writing that value (because of the previously described conversion between the names used in the specification and the names used in the code). For example, the **inputcode** relation will contain a pair composed of the variable value, `On`, and the code, `if(strcmp(compname = ‘‘On’’) ==0) return(ySwitch_on); else{printf(‘‘not a valid input value\n’’); return (-1);}`. This code is stored in relations because both the input and output routines must be generated at the node in the parse tree where the entire type definition occurs, not where the value itself occurs.

3.2. Code generation using rewrite rules

The generation of C code from an SCR specification using APTS rewrite rules is performed in three steps. First, a grammar is created that combines the language of the specification and the form of the corresponding C code. Second, relations are defined that capture the information in the specification. Finally, a set of transformations is defined that replaces the SCR specification with the corresponding C code.

The transformation-based code generator uses a grammar that combines both the form of an acceptable SCR specification and the form of the C code corresponding to the specification. A combined grammar is used so that during transformation, when the tree is a combination of SCR and C code, it is still a valid program. (This is a design decision that we made. During transformation APTS does not check that the parse tree remains valid, so the combined grammar is not strictly required by APTS.) The grammar used gives a parse tree where the nodes alternate between general structure and language-specific structure.

For example, consider the following general structure rule from the grammar:

```
case_ev = scr_case_ev | c_case_ev;
```

and its corresponding language-specific rules:

```
scr_case_ev = 'case' id case_branch_evs 'esac';
c_case_ev = case_branch_evs ;
```

A node denoting a case statement in an event table, `case_ev`, has only one child which is either a node denoting an SCR event table case statement, `scr_case_ev`, or a node denoting the C code corresponding to the event table case statement, `c_case_ev`. The SCR event table case statement is delimited by the keywords `'case'` and `'esac'` and includes the name of the identifier whose value the branches are conditioned on. It is also defined in terms of the general structure node `case_branch_evs`, which denotes the branches of the case statement. The C language-specific node is also defined in terms of `case_branch_evs`, which, in turn, is defined in terms of language-specific nodes denoting the branches. When the parse tree is initially created from an SCR specification, it contains general structure nodes alternating with SCR structure nodes. During the transformation process, SCR nodes are replaced by C nodes, so that at all times during the transformation, we have a valid parse tree.

Once an input SCR specification is parsed using this grammar, the relational database inference engine is called. In addition to the relations described in the preceding section, this code generator needs a new relation checking that the input is a valid SCR specification. This is necessary because the layered nature of the grammar allows an input file containing a mix of SCR and C code to be accepted by APTS as being syntactically valid. We check that only SCR nodes are used in alternation with the general structure nodes.

After the input specification has been parsed and all necessary relations have been calculated, the SCR specification is transformed into C code. The translation is done in several stages and the order of these stages matters because the transformations change the parse tree and thus may cause matches for later transformations to fail.

The first stage of transformation eliminates some of the dead code in the specification. The dead code on line 57 of the SIS example would be removed by the rewrite rule `if2` below. The match condition matches lists of if-statements where the first member .y is an if-statement and the second member .x is a list of if-statements. Recall that the node corresponding to line 57 has already been added to the **deadcode** relation during the building of the relational database, so the condition `deadcode(.y)` will be true. The complete list is replaced in the parse tree by the second component, eliminating the dead code branch.

```
if2: match(%scr_if_stmt_bodies,  .y .x%) | deadcode(.y)->
  rewrite(%scr_if_stmt_bodies, .x%);
```

The second step replaces enumerated values appearing in constant definitions and variable declarations with their new type-specific names. For example, the rule below replaces

TooLow on line 17 of the SIS example with type_mcPressure_TooLow. The relation **newname** contains the type-specific name associated with the identifier.

```
enum3: match(%var_decl, .x : .t, initially .y;%) |
  newname(.y,.z) ->
  rewrite(%var_decl, .x : .t, initially .z;%);
```

The third stage of the transformation converts most of the SCR language into C code. For example, the rule below replaces the type definition on line 5 of the SIS example with the corresponding C code given in Section 2.3. The relation rangefun contains the name to use for the range checking function in the variable .s.

```
typebody2: match(%scr_type_body, .x : integer in
  [.y,.z];%) | rangefun(.x,.s) -> rewrite(%c_type_body,
  # define .x int void .s (char * name, int value) {if
  ((value < .y) OR (value > .z)) {printf("value out of
  range:"); printf(strformout, name);}}%);
```

After this step is complete, the enumerated values remaining in expressions are replaced with their type-specific counterparts. Following this, the event operators are replaced with equivalent logical expressions. Finally, each primed expression is replaced with the name of the corresponding primed variable if the expression is a variable. If the primed expression is an enumerated value or an integer, then the prime is eliminated. The following are some of the rewrite rules used to perform these transformations.

```
equal: match(%expr, .x = .y%) |true ->
  rewrite(%expr, .x == .y%);
event1: match(%expr, @T(.x)%) |true ->
  rewrite(%expr, ((.x)') AND NOT(.x) %);
prime2: match(%expr, (.x)'%) | primename(.x,.y) ->
  rewrite(%expr, .y%);
prime11: match(%expr, (.x == .y)'%) | true ->
  rewrite(%expr, (.x)' == (.y)'%);
prime18: match(%expr, (.x)'%) | enumval(.x) or isint(.x)
  or const(.x) -> rewrite(%expr, .x%);
```

Each rule above replaces an SCR expression with an equivalent C expression. For example, for the SCR event expression @T(x=5), the C code is generated as follows. First, the node containing this expression is rewritten as @T(x==5) using the equal rule. Then, using event1, the expression is transformed into ((x==5)′) AND NOT(x==5). Next, prime11 and prime18 are used to place the prime in the correct location, rewriting the expression first as ((x)′==(5)′) AND NOT(x==5) and then as ((x)′==5) AND NOT(x==5). Finally, using prime2, the expression is rewritten as (prime_x==5) AND NOT(x==5).

3.3. Code generation using an accumulating relation

An alternative form of code generation relies solely on relations. Instead of transforming the source code into the target language, the target language code is accumulated in a relation. This approach keeps the two languages separate and preserves the original parse tree. On the negative side, it requires a great deal of additional calculation of relations. The relations used in this method are all but one of those used in the transformation-based method as well as several additional relations to hold the generated code and a relation to calculate the primed version of any expression.

Because the purely relational framework keeps the grammars for SCR and C separate, the parse tree contains only the productions for SCR language constructs. There is no need for the alternating style used in the parse tree for transformation-based code generation. Additionally, there is no need for the relation that checks that the input specification is pure SCR (rather than a mix of SCR and C). Because the grammars for SCR and C are separated, the input is only accepted by the parser if it is a valid specification in the SCR grammar. Although it is no longer combined with the SCR grammar, the C grammar is still included in the APTS grammar specification because it is used to structure the C code kept in the accumulating relation.

In this framework, additional relations perform the work done by the transformations in the other approach. A relation **prime** is used to calculate the primed C version of each expression so that if the primed form is needed during code generation, it will be available. The following are some of the rules for calculating **prime**. The first rule states that the prime of an identifier is the corresponding primed identifier, stored in relation **primename**. The second states that the prime of an integer or a constant is just that integer or constant. Finally, the third rule states that the prime of an equality expression involving two values is an equality expression of the primes of those two values. Note that in this inference rule the '=' used by SCR is replaced by the '==' used by C.

```
match(%expr, .x%) | primename(.x,.y) ->
  prime(loc(),%expr,.y%);

match(%expr, .x%) | isint(.x) or const(.x) ->
  prime(loc(),%expr,.x%);

match(%expr, .x = .y%) | prime(.x,.primex) and
  prime(.y,.primey)
  -> prime(loc(),%expr,.primex == .primey%);
```

All generated code is also held in relations. The execution code is developed in relations as previously described. The code generated by rewrite rules in the transformational approach is, in this approach, also placed in a relation. For each node, the accumulating relation stores the C code corresponding to the portion of the SCR specification represented by the subtree rooted at that node. This code is determined by the structure of the tree at that node and the code associated with the node's children. The code for the entire program is associated with the root of the tree.

The relational database rules listed below perform the same functions as the rewrite rules `if2`, `enum3`, and `typebody2` described in Section 3.2. The first rule eliminates the same dead code as rewrite rule `if2` by keeping as the code for the matched node only the code associated with the pattern variable .x. The second rule uses the type-specific name of an enumerated value as the C code to be generated for all expressions equivalent to that value, including the variable declarations handled in the transformation-based code generator by the rewrite rule `enum3`. The third relational database rule handles the same situation as rewrite rule `typebody2`, associating an SCR range type definition with the corresponding C code in relation **c_code**.

```
match(%if_stmt_bodies,  .y .x%) | deadcode(.y) and
  c_code(.x,.codex)and not(deadcode(loc()))->
  c_code(loc(),%c_if_stmt_bodies, .codex%);

match(%expr, .x%) | enumval(.x) and newname(.x,.y) ->
  c_code(loc(),%expr,.y%) ;

match(%type_body, .x : integer in [.y,.z];%) |
  rangefun(.x,.s)-> c_code(loc(),%c_type_body,
  # define .x int void .s (char * name, int value)
  {if ((value < .y) OR (value > .z)) {printf("value
  out of range:");printf(strformout, name);}}%);
```

Below are several rules for generating C code for event expressions and other simpler expressions. The first two rules state that variables and integers in the SCR specification are not changed in the C code. The third rule calculates the C code for an expression that checks the equivalence of two expressions by combining the previously calculated code for each of the subexpressions. The last rule calculates the C code for the “at true” event, converting it into an equivalent logical expression. Note that this rule assumes that the code corresponding to the prime value of the expression has already been calculated and is stored in `.primex`.

```
match(%expr, .x%) | not(enumval(.x)) and isavar(.x)
  -> c_code(loc(),%expr,.x%);

match(%expr, .x%) | isint(.x) -> c_code(loc(),%expr,.x%);

match(%expr, .x = .y%) | c_code(.x,.codex) and
  c_code(.y,.codey) ->
  c_code(loc(),%expr, .codex == .codey%) ;

match(%expr, @T(.x)%) |prime(.x,.primex) and
  c_code(.x,.codex) ->
  c_code(loc(),%expr, (.primex) AND NOT(.codex) %) ;
```

Consider again the example `@T(x=5)`. Using the relational database rules above and those given earlier for the relation **prime**, we can calculate the relations **c_code** and **prime** for each of the subcomponents. For example, using these rules, `prime(5) = 5`, `prime(x) = prime_x`, `c_code(5) = 5`, and `c_code(x) = x`. Using these as a basis, we determine the values of the relation for `x=5`, namely, `prime(x=5) = prime_x ==5` and `c_code(x=5) = x==5`. Now, we can calculate the value of `c_code` for the full expression: `c_code(@T(x=5)) = (prime_x==5) AND NOT (x==5)`.

4. Discussion

Given an SCR specification, the transformation-based and relation-based strategies generate exactly the same code. For the SIS example, which generated 293 lines of C code, the transformation-based strategy required four minutes and the relation-based strategy required 20 minutes.[5] We also used both generators to generate code from the SCR requirements specification for a cryptographic device [25]. This latter specification contains 36 variables and 20 function definitions and is 658 lines long. The translation of the SCR tabular specification into the SCR grammar used by APTS was done by hand, although such a translation could be automated. In the second example, the code generator using rewrite rules required approximately 12 hours to generate 2028 lines of C code. The generator that accumulated code in a relation required far longer—more than 72 hours. We observed that the version using rewrite rules spent most of the time building the relational database. Clearly, a speedup of the APTS relational database inference engine would greatly improve the execution times for both code generators.

For our purposes, APTS was a useful tool for exploring the two strategies since it contained the means to implement both the relational and transformation-based approaches. Clearly, the time needed to generate code using APTS is currently too great to use APTS to construct a production-quality system. Paige had planned a number of improvements to APTS [34], which he estimated would improve the translation rate by a factor of 6000. These improvements included translating the SETL2 code of APTS into C (for an expected speedup factor of 30) and using partial evaluation to convert the APTS interpreters into compilers (for an expected speedup factor of 10). If these two improvements were made, the times for generating code for SIS would be in seconds instead of minutes and for the cryptographic device would be in minutes instead of hours.

While the transformation-based approach generates code more quickly in APTS, the relation-based strategy is more straightforward. Though many of the rewrite rules are easily understood because they relate directly to the inference rules used by the relation-based strategy, the necessary execution ordering of the rewrite rules is less intuitive. It is also important to note that a purely transformational strategy was impossible because code sometimes needed to be calculated at one point in the parse tree and generated at a different point in the tree. In the generator using rewrite rules, this was done by placing the code in a relation in the relational database and passing it to the transformational engine.

Changing the target language from C to some other language (or modifying the C code to be generated) would require approximately the same amount of work for both generators. Most relations used by both strategies refer only to the SCR specification and thus would not change. For both code generators, the grammar used by APTS would require modification. With the transformation-based strategy, the C language structures in the interleaved grammar would be replaced by the language structures of the new target language. With the relation-based strategy (or if the translation-based strategy did not use an interleaved grammar), changing the grammar is even easier. Since the new language need not be interleaved with the SCR grammar, it can simply be added to the grammar file in place of the C grammar. Finally, the transformations or the accumulating relation must be modified. In both cases, the actual conditions for the rules (rewrite or inference) remain the same. What changes is the result of the rewrite rule or the value stored in the accumulating relation.

It should also be noted that code can be generated for incomplete specifications. In particular, code can be generated for partial specifications in which all of the dependent variables have not yet been defined by a table function. Because the code generated for each function definition is independent of the code generated for any other function definition, it is possible to generate code separately for each function in the specification. However, the variable declarations, type definitions, and constant definitions for any variables, types, and constants used in the table need to be included in the partial specification in order for code to be properly generated for the table.

The generated C code performs very well. The code for the communications device processed an input file with 17 monitored events in less than one second. We have no hand-written C code to which the generated code can be compared, but the SCR toolset has a simulator [17] that produces Java code to simulate the behavior of the state machine defined by the specification. Our C code runs faster than the simulator's Java code, but a fair comparison of the two is difficult. Java code is generally slower than C code and the simulator also uses a GUI interface, slowing the running time even more. The simulator has one advantage over our C code; it has been optimized to update a variable only when at least one of the variables on which it depends has been changed.

A major contribution of Paige's research is finite differencing. Although the APTS reference manual [33] states that APTS contains techniques for optimizing the generated code, how to use these techniques within APTS is not documented. However, some preliminary work on how code generated from SCR specifications could be optimized has been done. One serious source of inefficiency in the generated code is that each new input requires an update to every variable in the program. One obvious way to reduce this inefficiency, which is used by the SCR simulator, is to use the variable dependencies (computed automatically by the SCR toolset) to determine which variables could potentially change value when a given input variable changes value and eliminate updates to the remaining variables. Information from invariants may also be used to further optimize the variable updates [23].

Other ways to reduce inefficiency are to identify parts of the specification that lead to dead code and to redundant code and omit code generation for those parts. Our implementations currently only eliminate the obviously dead code—branches labeled by `false` and `never`. State invariants constructed using the algorithms described in [21, 24] can be used to identify parts of the specification that lead to dead code or to redundant code [22]. For example, [21] shows that $\texttt{tOverridden} = \textit{true} \rightarrow \texttt{mcPressure} \neq \texttt{High}$ is a state invariant of the SIS specification. This invariant implies that `tOverridden` cannot change from *true* to *false* if `mcPressure = High` in the old state. Hence, the disjunct on line 46 of the SIS specification `@T(NOT(mcPressure = High))` may be ignored because it produces dead code. Similarly, in the specification of an automobile cruise control system, one of the automatically generated state invariants is $M = \texttt{Inactive} \rightarrow \texttt{IgnOn}$ [21]. Hence, the event `@T(Lever=const)` WHEN ($M = \texttt{Inactive}$ AND `EngRunning` AND NOT `Brake` AND `IgnOn`) may be replaced by the equivalent event `@T(Lever=const)` WHEN ($M = \texttt{Inactive}$ AND `EngRunning` AND NOT `Brake`).

5. Related work

Generating code from specifications is not a new idea. The APTS translators for SETL2 and SQ2+ [7] can be used to translate specifications in these high-level languages into C. Like APTS, META-AMPHION [28], REFINE [37], and KIDS [39] may be used to design translators from high-level declarative specifications into executable programs. Moreover, several commercial tools generate code from specifications. For example, Statemate [13] generates C or Ada code from Statechart specifications and Telelogic's SCADE generates C or Ada code from LUSTRE [12]. Also, a C++ code generator for specifications written in RSML is discussed in [14]. In [41], C-like imperative code is generated from specifications given in E-FRP (Event-Driven Functional Reactive Programming) and transformations are applied to optimize the code. As in SCR, E-FRP variables only change value in response to events. Interest has been expressed in translating E-FRP into SCR [41].

Our two strategies share similarities with previous code generation methods. Both store additional information in relations. Many (but not all) of these relations are defined over nodes in the parse tree, making those relations similar to the attributes used in attribute grammar systems [26], such as the Synthesizer Generator [38]. Like an attribute grammar, our relational approach treats the target code as a synthesized attribute of the parse tree for the specification.

Our use of APTS rewrite rules to generate code is similar to Cai and Paige's [7] use of APTS to translate SETL2 and SQ2+ into C. Their translations used rewrite rules to generate the code, just as our implementation did. One difference between our work and theirs is that they also made use of APTS built-in finite differencing and dominated convergence optimizations, while we did not.

Our transformational strategy is also similar to the HATS transformational programming system [42] in its use of tree rewriting rules. Both HATS and our transformational strategy use rewrite rules that modify the actual tree to hold the changed code, and both require that the transformations always produce valid trees. Both also condition the rewrite rules on the matching of patterns in the trees. One difference is that our APTS-based transformational strategy also allows the relations to hold additional information and to be used as conditions for matching in rewrite rules. Another difference between HATS and our approach is that HATS may sometimes use problem-specific transformations, which our transformational system does not currently support.

Two other systems, Twig [1] and iburg [11], produce code generators that modify the parse tree. Unlike APTS, which makes many passes over the parse tree, these code generator generators work by making only two passes over the parse tree. The first pass finds a set of minimal cost patterns that cover the tree. The second pass executes the semantic actions associated with these patterns. Twig and iburg do not replace the code in the tree with target language code as our transformational system does. Instead, a pattern is matched, code associated with the pattern is generated to a file, and then the tree is reduced using the rewrite rule for the pattern. This process is repeated until the whole tree has been reduced.

As noted in Section 2.3, the code generated by our code generators uses an execution model similar to the execution model of SCR systems used in the translation of SCR into Promela [3]. Both use two sets of variables, one for the old state values and one for the new state values. Both encode the function tables as conditional statements in the target language and both execute the code for the functions in an order determined by the dependency relationship on the variables. One difference is that the Promela translation uses nondeterministic choice to implement the branches in a table, which is impossible in C. This is not a significant difference (i.e., it does not result in a possibly different semantics), since the conditions on the branches of a table are required to be disjoint [18], a requirement that is verified by the consistency checker in the SCR toolset.

6. Conclusion and future work

This paper described our experiments in developing code generators using APTS. Two different strategies to generate C code from SCR requirements were implemented. One strategy transforms a parse tree in the specification language into a parse tree in the target language, while the other accumulates the generated code in a relation associated with nodes in the specification language parse tree. Both APTS implementations generate the same code, and both perform a significant amount of analysis before generating code. Though APTS is currently too slow to be used as part of a production-quality system, it is likely that implementing the improvements that Paige suggested would lead to a system that uses a relational database and rewrite rules to generate code at an acceptable speed.

In the future we plan to study the problem of certifying that the generated code is correct, possibly via translation validation [36]. We also will apply optimizations, such as those described in Section 4, to SCR specifications.

Appendix A: Simple abstract grammer for SCR

The abstract grammar for SCR presented in this section is a variation of that given by Bharadwaj and Sims [4]. The rules are given in the form of a Regular Right Part (RRP) grammar [27]. As discussed in Section 2.1, some rules are replaced by equivalent BNF grammar rules in the grammars used by APTS.

In the grammar rules below, boldface type is used to indicate tokens in the language. Italics denotes nonterminals. The metasymbol $(\cdots)^*$ represents 0 or more occurrences of what is inside the parentheses. Similarly, $(\cdots)^+$ means 1 or more occurrences of what is inside the parentheses. Instances of parentheses without the * or $^+$ should be read as actual occurrences of parentheses in the language, i.e., tokens. A vertical bar, |, is used to denote choice.

Formally, an SCR specification begins with the keyword **spec** and an identifier name for the specification (line 1 in the example). This is followed by the six sections of the specification.

spec ::= **spec** *id type_defs constant_defs var_declarations*
assumptions assertions function_defs

Type definitions. Users can define their own types in SCR. Two kinds of type definitions are allowed, an integer range and a list of enumerated values. In addition to these user-defined types, there are two built-in types, integers and booleans.

type_defs ::= **type definitions** (*id*: *ud_type*;)*
ud_type ::= **integer in [***sint*, *sint***]**
| **enum in** { *id* (, *id*)*}
type ::= *id*
| *built_in_type*
built_in_type ::= **integer**
| **boolean**

The SIS example in Section 2.2 contains three user-defined types (lines 2–5).

Constants. Constants are defined by setting the identifier equal to some expression and declaring the type.

constant_defs ::= **constant definitions** (*id* = *expr*: *type*;)*

There are two integer constants in the SIS example, `Low` and `Permit` (lines 6–8).

Variables. There are four categories of variable declarations, corresponding to the four types of variables in SCR specifications: monitored, controlled, term, and mode classes. Each variable is either given an initial value or a "-" is placed in the declaration, signifying that no initial value is specified.

var_declarations ::= **monitored variables** (*var_decls*;)*
controlled variables (*var_decls*;)*
term variables (*var_decls*;)*
mode classes (*var_decls*;)*
var_decls ::= *id* (, *id*)*: *type*, **initially** *init_val*
init_val ::= *expr*
| **-**

The variable declarations for the SIS example appear on lines 9–17.

Assumptions. The variable declarations are followed by the section containing assumptions about the environment. Each assumption is given a name and is defined by a predicate. Predicates may contain events and expressions involving one or two states. A two-state expression is one that may use the new value of a variable, denoted x', as well as its old value, x. The syntax for assertions is the same as the syntax for assumptions.

assumptions ::= **assumptions** (*id*: *predicate*;)*
assertions ::= **assertions** (*id*: *predicate*;)*

predicate ::= *event*

| *bool_expr_prime*

| *predicate logical_op predicate*

| (*predicate*)

The SIS example contains one assumption and no assertions. These two sections can be found in lines 18–22.

Function definitions. The function definition section contains the functions used to update the values of variables. In a complete specification, there should be a function for each dependent variable. Two types of tables exist for specifying the update functions, condition tables and event tables. The start of a new table is indicated by the keyword **var** followed by the name of the variable that the table defines. After the variable name comes punctuation indicating whether the variable will be defined by a condition table (**==**) or an event table (**:=**).

Moded condition tables (`cSafety_Injection`, lines 48–60) are expressed using a case-statement. The identifier following the keyword **case** is the name of a mode class variable. This is followed by a set of branch statements, each of which begins with a list of identifiers that are modes of the mode class variable. The semantics selects the branch containing the mode that is the current value of the mode class variable named in the case-statement. The if-statement corresponding to this branch is executed. An if-statement contains pairs of boolean expressions and expressions joined by an arrow. B −> E means that if the boolean expression B evaluates to true, then the variable is assigned the value of the expression E.

Event tables are defined similarly to condition tables (see `mcPressure`, lines 24–39). The only difference is that in place of if-statements, event tables use event statements. An event-statement contains pairs of events and two state-expressions joined by an arrow. The semantics of Ev −> E is that if the event Ev evaluates to true, then the variable is assigned the value of the expression E. If none of the events evaluates to true, then the value of the variable is not changed.

Unmoded event and condition tables can also be defined. The form of an unmoded table is the same as one branch of a moded table. The variable `tOverridden` has its update function defined by an unmoded event table (lines 40–47).

function_defs ::= **function definitions** (*funct_def*)*

funct_def ::= **var** *id* (**==** *cond_tab* | **:=** *event_tab*)

cond_tab ::= *case_if*

| *if_stmt*

case_if ::= **case** *id* (*case_branch_if*)$^+$ **esac**

case_branch_if ::= **[]** *id* (**,** *id*)**if_stmt*

if_stmt ::= **if** (**[]** *bool_expr* −> *expr*)$^+$ **fi**

event_tab ::= *case_ev*

| *ev_stmt*

case_ev ::= **case** *id* (*case_branch_ev*)$^+$ **esac**

case_branch_ev ::= **[]** *id* (**,** *id*)**ev_stmt*

ev_stmt ::= **ev** (**[]** *eventp* −> *expr_prime*)$^+$ **ve**

Events. There are three basic types of events.@**T**(B) is true when the value of the expression B changes from false to true. That is, @**T**(B) evaluates the boolean expression B in the old state and the new state and, if B is false in the old state and true in the new state, then @**T**(B) is true. Otherwise, it is false. Similarly, the event @**F**(B) is true when the value of the boolean expression B changes from true to false. The event @**C**(E) is true whenever the value of the expression E in the old state is different from its value in the new state. Notice that E can be any expression, not just one with a boolean value.

The basic events can be combined with the **WHEN** and **WHENP** operators to form conditional events. The conditional event A **WHEN** B is true when the basic event A is true and the boolean expression B is true. **WHENP** is similar to **WHEN** except that the boolean expression following **WHENP** is allowed to include primed variables.

Logical combinations of events are also allowed. Finally there is a special event, **never**, whose presence in an event table indicates that a particular situation cannot occur.

basic_event ::= @**T**(*bool_expr*)
| @**F**(*bool_expr*)
| @**C**(*expr*)
cond_event ::= *basic_event* **WHEN** *bool_expr*
| *basic_event* **WHENP** *bool_expr_prime*
event ::= *basic_event*
| *cond_event*
| *event logical_op event*
eventp ::= *event*
| **never**

Expressions. There are two categories of expressions, one-state expressions which describe the values of variables in a single state, and two-state expressions which may include the values of variables in both the old and new states. Each category of expressions is divided into three types based on the type of result they return—arithmetic, enumerated, and boolean. Arithmetic expressions include signed integers, identifiers that have arithmetic type, the standard binary operations on arithmetic expressions, and the unary plus and minus operations. Enumerated expressions are just identifiers having an enumerated type. (This includes the possible values of an enumerated type.) Boolean expressions consist of true and false, identifiers of boolean type, and the standard logical operations applied to boolean expressions. Boolean expressions also include equality checks between pairs of boolean expressions and pairs of enumerated expressions. All of the basic arithmetic relational operators are allowed in boolean expressions.

arith_expr ::= *sint*
| *id*
| *arith_expr arith_op arith_expr*
| −*arith_expr*
| +*arith_expr*
| (*arith_expr*)

```
enum_expr ::= id
bool_expr  ::= bool
             | id
             | NOT bool_expr
             | bool_expr logical_op bool_expr
             | bool_expr = bool_expr
             | (bool_expr)
             | enum_expr = enum_expr
             | arith_expr rel_op arith_expr
expr       ::= bool_expr
             | arith_expr
             | enum_expr
```

The grammar for two-state expressions is similar to that of one state expressions. The only difference is that primed identifiers, referring to the new state value of the identifier, are allowed in these expressions.

```
arith_expr_prime  ::= id'
                    | arith_expr
                    | arith_expr_prime arith_op arith_expr_prime
                    | −arith_expr_prime
                    | +arith_expr_prime
                    | (arith_expr_prime)
enum_expr_prime ::= id'
                    | enum_expr
bool_expr_prime  ::= id'
                    | bool_expr
                    | NOT bool_expr_prime
                    | bool_expr_prime logical_op bool_expr_prime
                    | arith_expr_prime rel_op arith_expr_prime
                    | (bool_expr_prime)
                    | enum_expr_prime = enum_expr_prime
                    | bool_expr_prime = bool_expr_prime
expr_prime         ::= bool_expr_prime
                    | arith_expr_prime
                    | enum_expr_prime
```

Lexical categories. Finally, there are some productions for the basic lexical categories. Signed integers and the boolean values, true and false, are defined as well as all the standard relational and arithmetic operators. Note that the logical operators AND and OR can also be written as && and || respectively.

```
rel_op      ::= = | != | > | < | >= | <=
logical_op ::= AND | OR | && | ||
arith_op    ::= + | − | ∗ | /
```

$$\begin{array}{lll}
\textit{bool} & ::= & \textbf{true} \\
 & & \mid \textbf{false} \\
\textit{digit} & ::= & \textbf{0} \mid \textbf{1} \mid \textbf{2} \mid \textbf{3} \mid \textbf{4} \mid \textbf{5} \mid \textbf{6} \mid \textbf{7} \mid \textbf{8} \mid \textbf{9} \\
\textit{int} & ::= & (\textit{digit})^{+} \\
\textit{sint} & ::= & -\textit{int} \mid \textit{int} \\
\textit{alpha} & ::= & \textbf{a} \mid \ldots \mid \textbf{z} \mid \textbf{A} \mid \ldots \mid \textbf{Z} \\
\textit{char} & ::= & \textit{alpha} \\
 & & \mid \textit{digit} \\
 & & \mid \_ \\
\textit{id} & ::= & \textit{alpha}(\textit{char})^{*}
\end{array}$$

Appendix B: Specifying a simple control system in SCR

This appendix contains the tabular forms of the function definitions for the Safety Injection System (SIS) example given in Section 2.2. The mode class `mcPressure` (see lines 24–39 in the SIS example) is defined by the mode transition table (a special class of event table) shown in Table 1.

The term `tOverridden` is defined by an unmoded event table (see lines 40–47 of the example). The corresponding tabular form is shown in Table 2.

Table 1. Mode transition table for `mcPressure`.

Old mode	Event	New mode
TooLow	@T(mWaterPres ≥ Low)	Permitted
Permitted	@T(mWaterPres ≥ Permit)	High
Permitted	@T(mWaterPres < Low)	TooLow
High	@T(mWaterPres < Permit)	Permitted

Table 2. Event table for `tOverridden`.

Variable	Events	
	@T(mBlock = On) WHEN (mReset = Off AND NOT(mcPressure = High))	@T(mcPressure = High) OR @T(NOT(mcPressure = High)) OR @T(mReset = On) WHEN NOT(mcPressure = High)
tOverridden'	True	False

Table 3. Condition table for `cSafety_Injection`.

Mode	Conditions	
High, Permitted	True	False
TooLow	tOverridden	NOT tOverridden
cSafety_Injection	Off	On

The controlled variable `cSafety_Injection` is defined by a moded condition table (see lines 48–60). Table 3 shows the tabular form of this definition.

Acknowledgments

We wish to acknowledge Bob Paige's work on the APTS system and the assistance he provided us in using APTS. Bob provided us with a first version of the SCR grammar for APTS and also answered many of our questions about the system. Much of the initial work on the SCR abstract grammar was done by Ramesh Bharadwaj. Discussions with Myla Archer were also very useful in our development of the grammar. The algorithms for constructing invariants from SCR specifications and the idea of using invariants to identify parts of the specification that would lead to redundant or dead code are due to Ralph Jeffords. We thank the anonymous reviewers and our colleagues Myla Archer and Ralph Jeffords for helpful comments on drafts of this paper and Annie Liu and Scott Stoller for helpful discussions.

Notes

1. By convention, the names of monitored variables begin with 'm', of controlled variables begin with 'c', of terms begin with 't', and of mode classes begin with 'mc'. The names of user-defined types begin with 'y', and the names of types associated with mode classes begin with 'type_'.
2. In SCR specifications, the initial value of a dependent variable can often be derived from the initial values of variables upon which the variable depends.
3. The entry `false` in the table defining `cSafety_Injection` (see Table 3 in Appendix B) is an artifact of the tabular format. It means that `cSafety_Injection` is never equal to `ON` when the mode is `High` or `Permitted` and would therefore correspond to dead code.
4. The conversion is currently done by hand but would be easy to automate.
5. Execution times are for a Sun Ultra 450 with 2 UltraSPARC-II 296 MHz CPUs and 2 GB memory, running Solaris 5.6.

References

1. Aho, A.V., Ganapathi, M., and Tjiang, S.W.K. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, **11**(4) (1989) 491–516.
2. Archer, M. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, **29**(1–4) (2001) 139–181.
3. Bharadwaj, R. and Heitmeyer, C. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, **6**(1) (1999) 37–68.
4. Bharadwaj, R. and Sims, S. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '2000)*, S. Grat and M. Schwartzbach (Eds.). LNCS 1785, Berlin, 2000, pp. 378–399.
5. Blackburn, M., Knickerbocker, R., and Kasuda, R. Applying the test automation framework to the Mars Lander Touchdown Monitor. In *Lockheed Martin Joint Symposium*, 2001.
6. Cai, J. and Paige, R. Program derivation by fixed point computation. *Science of Computer Programming*, **11** (1988/89) 197–261.
7. Cai, J. and Paige, R. Towards increased productivity of algorithm implementation. *Proceedings ACM SIGSOFT 1993, Software Engineering Notes*, **18**(5) (1993) 71–78.

8. Courtois, P.-J. and Parnas, D.L. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*. Baltimore, MD, 1993, pp. 315–323.
9. Easterbrook, S., Lutz, R., Covington, R., Ampo, Y., and Hamilton, D. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, **24**(1) (1998) 4–14.
10. Faulk, S.R., Finneran, L., Kirby, Jr., J., Shah, S., and Sutton, J. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conf. on Computer Assurance (COMPASS '94)*. IEEE Press, Gaithersburg, MD, 1994, pp. 3–8.
11. Fraser, C.W., Hanson, D.R., and Proebsting, T.A. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, **1**(3) (1992) 213–226.
12. Halbwachs, N., Raymond, P., and Ratel, C. Generating efficient code from data-flow programs. In *Third Intern. Symposium on Programming Language Implementation and Logic Programming*. Passau (Germany), 1991, pp. 207–218.
13. Harel, D. et al. Statemate: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Eng.*, **SE-16**(4) (1990) 404–414.
14. Heimdahl, M.P.E. and Keenan, D.J. Generating code from hierarchical state-based requirements. In *Proc. of the IEEE International Symposium on Requirements Engineering*, 1997, pp. 450–467.
15. Heitmeyer, C. Software cost reduction. In *Encyclopedia of Software Engineering*, J.J. Marciniak (Ed.). (Second edition), John Wiley & Sons, Inc., New York, NY, 2002, pp. 1374–1380.
16. Heitmeyer, C., Kirby, J., Labaw, B., Archer, M., and Bharadwaj, R. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, **24**(11) (1998), 927–948.
17. Heitmeyer, C., Kirby, Jr., J., Labaw, B., and Bharadwaj, R. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*. A.J. Hua and M.Y. Vardi (Eds.). LNCS 1427, Vancouver, Canada, 1998, pp. 526–531.
18. Heitmeyer, C.L., Jeffords, R.D., and Labaw, B.G. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, **5**(3) (1996) 231–261.
19. Heninger, K., Parnas, D.L., Shore, J.E., and Kallander, J.W. Software requirements for the A-7E aircraft. Technical Report 3876, NRL, Washington, DC, 1978.
20. Hester, S.D., Parnas, D.L., and Utter, D.F. Using documentation as a software design medium. *Bell System Tech. J.*, **60**(8) (1981) 1941–1977.
21. Jeffords, R. and Heitmeyer, C. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, Nov. 3–5, Lake Buena Vista, FL, 1998, pp. 56–69.
22. Jeffords, R.D. Personal communication, 2001.
23. Jeffords, R.D. Personal communication, 2002.
24. Jeffords, R.D. and Heitmeyer, C.L. An algorithm for strengthening state invariants generated from requirements specifications. In *Proc. Fifth IEEE International Symposium on Requirements Engineering*, 2001, pp. 182–191.
25. Kirby, Jr., J., Archer, M., and Heitmeyer, C. SCR: A practical approach to building a high assurance COMSEC system. In *Proc. 15th Annual Computer Security Applications Conference (ACSAC '99)*, 1999, pp. 109–118.
26. Knuth, D.E. Semantics of context-free languages. *Mathematical Systems Theory*, **2**(2) (1968) 127–145.
27. LaLonde, W. Regular right part grammars and their parsers. *Communications of the ACM*, **20**(10) (1977) 731–741.
28. Lowry, M.R. and Baalen, J.V.V. META-AMPHION: Synthesis of efficient domain-specific program synthesis systems. *Automated Software Engineering*, **4** (1997) 199–241.
29. Meyers, S. and White, S. Software requirements methodology and tool study for A6-E technology transfer. Technical report, Grumman Aerospace Corp., Bethpage, NY, 1983.
30. Miller, S. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd ACM Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998, pp. 44–53.
31. Paige, R. Programming with invariants. *IEEE Software*, **3**(1) (1986) 56–69.
32. Paige, R. Symbolic finite differencing—Part 1. In *Proc. ESOP 90*, N. Jones (Ed.), LNCS 432, 1990, pp. 36–56.
33. Paige, R. APTS external specification manual (rough draft). Unpublished manuscript, 1993. Available at http://www.cs.nyu.edu/ jessie/.

34. Paige, R. Viewing a program transformation system at work. In *Proc. Joint 6th Int'l Conf. on Programming Language Implementation and Logic Programming (PLILP) and 4th Int'l Conf. on Algebraic and Logic Programming (ALP)*. LNCS 844, 1994, pp. 5–24.
35. Parnas, D.L., Asmis, G., and Madey, J. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, **32**(2) (1991) 189–198.
36. Pnueli, A., Siegel, M., and Singerman, E. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1998)*. LNCS 1384, 1998, pp. 151–166.
37. Reasoning Systems. Refine User's Guide Version 3.0, 1990.
38. Reps, T.W. and Teitelbaum, T. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, NY, 1989.
39. Smith, D.R. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, **16** (1990) 1024–1043.
40. Snyder, K. The SETL2 programming language. Technical Report 490, Courant Institute/ New York University, 1990.
41. Wan, Z., Taha, W., and Hudak, P. Event-driven FRP. In *Proc. Fourth International Symposium on Practical Aspects of Declarative Languages (PADL02)*, LNCS 2257, 2002, pp. 155–172.
42. Winter, V.L., Kapur, D., and Berg, R.S. Refinement-based derivation of train controllers. In *High Integrity Software*, V.L. Winter and S. Bhattacharya (Eds.). Kluwer Academic Publishers, Norwell, MA, 2001, Ch. 9, pp. 197–240.