Siddhartha: A Method for Developing Domain-Specific Test Driver Generators

Arthur Alexander Reyes
Dept. Computer Science & Engineering
University of Texas
Arlington, TX 76019-9015 U.S.A.
reyes@cse.uta.edu

Debra J. Richardson
Dept. Information & Computer Science
University of California
Irvine, CA 92697-3425 U.S.A.
djr@ics.uci.edu

Abstract

Siddhartha applies the domain-specific language (DSL) paradigm to solve difficult problems in specification-based testing (SBT). Domain-specific test case data specifications (TestSpecs) and difficult-to-test program design styles engender difficult SBT problems, which are the essential phenomena of interest to Siddhartha. Difficult-to-test program design styles are explicitly represented by domain-specific, unit test driver reference designs that accommodate the problematic program design styles. DSLs are developed to represent both TestSpecs and Driver reference designs. A DSL language processing tool (a translator) is developed that maps TestSpecs into Drivers. We developed a prototype implementation of Siddhartha via Reasoning SDK (formerly known as Software Refinery) and developed two domain-specific TestSpec→Driver translators. Each translator generated Drivers that revealed new failures in a real-world digital flight control application program.

1. Introduction

Specification-based testing (SBT) [9] exploits formal specifications (Specs) as (1) unambiguous standards against which to automatically check tested program behavior and as (2) templates against which to automatically generate test case data specifications (TestSpecs). A computer program unit subjected to testing is referred to as a unit under test (UUT). An important class of SBT tool encompasses those that first generate TestSpecs and then refine them into unit test driver programs (Drivers). Examples of SBT tools in this class are ASTOOT [3], DAIST [4], Daistish [7], DAS-BOOT [12], and T-VEC [1]. Drivers control and observe UUT behavior via programmatic invocations. Drivers feature an embedded oracle to automatically check UUT test outcomes with respect to TestSpec.

Many SBT tools in this class are not modular and tightly couple TestSpec generation and TestSpec→Driver translation. Development of TestSpec→Driver translators is marginalized in most SBT research literature, even though these translators are required by every SBT tool in this class. Difficult TestSpec→Driver translation problems are often ignored or assumed away. Doing so produces SBT tools that can only be applied to UUTs that bear a strong resemblance to their Specs.

The UUT design process must balance several software quality attributes (e.g., correctness, efficiency, ease of analysis, maintainability, testability, etc.), each prioritized by domain-specific business needs. Thus domain-specific UUT design styles can emerge. For example, if run-time efficiency is a high priority, a design style may be characterized by ubiquitous reference and assignment to global variables. This is a difficult-to-test UUT design style. If Drivers generated by a SBT tool cannot accommodate such design styles, the SBT tool is difficult, if not impossible to apply in that application domain.

The application domain of digital avionics and flight control software is a good example of this problem. UUTs in this application domain have for a very long time been characterized by insufficient interface definition (i.e., too few formal parameters, ubiquitous global variables, side-effecting behavior) and hardware representation specifications for data structures. General purpose SBT tools are difficult to apply to these UUTs. In response, digital avionics and flight controls software vendors develop custom SBT tools that respect this UUT design style [13]. Unfortunately, little guidance for developing domain-specific SBT tools can be found in the SBT research literature.

This paper describes Siddhartha, a disciplined, example-driven, iterative, non intrusive, alternative method for developing domain-specific TestSpec→Driver translators. Domain-specific TestSpec→Driver translators provide essential

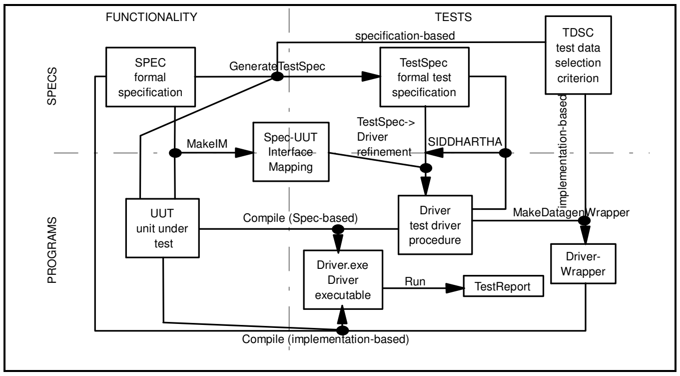

Figure 1: Generic domain model diagram

automation support for SBT in new application domains. Generated Drivers are expressed in design styles that specifically accommodate domain-specific, difficult-to-test UUT design styles.

Section 2 presents Siddhartha's conceptual model foundation. Section 3 describes a prototype implementation of Siddhartha via Reasoning SDK (formerly known as Software Refinery). Section 4 describes two domain-specific TestSpec→Driver translators we developed via the prototype implementation. Section 5 describes the results of evaluating these translators by replicating tests performed by the NASA Software Independent Verification and Validation (IV&V) Facility for a real-world digital flight controls application program.

2. Siddhartha's conceptual model

Siddhartha applies the domain-specific language (DSL) paradigm [8] to solve difficult problems in SBT. The DSL paradigm is (1) discover abstractions for essential phenomena in the application domain, (2) represent these abstractions via DSLs, and (3) develop DSL processing tools (e.g., parsers, printers, editors, type checkers, interpreters, translators, synthesizers, etc.) to provide essential, automatic problem-solving support.

Domain-specific TestSpec representations and difficult-to-test UUT design styles are the essential phenomena of interest to Siddhartha. An expert Test Engineer applying Siddhartha (1) discovers abstractions for TestSpec representations and difficult-to-test UUT design styles, (2) represents these abstractions via DSLs, and (3) develops a TestSpec→Driver translator to provide essential automation support for SBT. These activities are supported by a predefined domain model, process description, library of reusable tool components, and reference architecture.

Siddhartha's conceptual model is comprised of a predefined domain model and a process description. The domain model provides a starting point for domain analysis [10]. The process description tells how to develop the domain-specific TestSpec→Driver translator. Figure 1 presents Siddhartha's domain model diagram.

Nodes represent classes of software artifacts. Hyper edges (i.e., edges with potentially more than one tail) represent tools or processes that map artifacts at each tail of an edge to an artifact at the head of an edge.

As indicated in the diagram, Siddhartha assumes the existence of computer program units under test (UUTs), formal specifications of each UUT's intended behavior (Specs), formal test specifications (TestSpecs) generated from Spec and/or UUT in accordance with a test data selection criterion (TDSC), unit test drivers (Drivers), and reports of test results (TestReports). Siddhartha also assumes the existence of tools that generate TestSpecs (GenerateTestSpec), compile Drivers into executables (Compile), and invoke Drivers (Run). Depending on the application domain, two object classes (and their associated tools or processes) are optional: Spec→UUT Interface Mappings and Driver-Wrappers. Spec→UUT Interface Mappings permit representational flexibility in corresponding Spec and UUT data and control structures. Driver-Wrappers invoke Drivers with selected test case input data.

The goal of applying Siddhartha is to produce a TestSpec→Driver translator. The Siddhartha method itself is denoted by the hyper edge that takes as input a general, example TestSpec and manually-coded Driver and produces as output a TestSpec→Driver translator.

An expert Test Engineer applying Siddhartha follows a process description with four phases: domain analysis, example selection, translator development, and translator testing and deployment.

In the domain analysis phase, the Test Engineer maps nodes and hyper edges in the generic domain model to real-world software artifacts, tools, and processes in the unit testing application domain. Doing so encourages the Test Engineer to search for already-used representations and tools that can be exploited to support automatic TestSpec→Driver translation.

In the example selection phase, the Test Engineer finds a small collection of general, example TestSpecs. For each example TestSpec, the Test Engineer manually codes one or more Drivers. Each Driver is expressed in a different reference design. Each reference design must correctly implement the TestSpec and accommodate any domain-specific (and UUT programming language-specific), difficult-to-test UUT design styles. The Test Engineer evaluates the manually-coded Drivers and selects the Drivers expressed in the most promising reference design. This yields a collection of example (TestSpec, Driver) pairs. These example pairs are test cases for the TestSpec→Driver translator under development.

During TestSpec→Driver translator development, the Test Engineer designs a (preferably recursive) translation function that maps TestSpec fragments into corresponding Driver reference design fragments. Because Siddhartha treats both TestSpecs and Driver reference designs as DSLs, this design activity modularizes both the TestSpec DSL and Driver reference design DSL into grammar productions (i.e., it effectively optimizes the representations of both TestSpec DSL and Driver reference design DSL to support efficient TestSpec→Driver translation).

Based on the modularization imposed by the translator design, the Test Engineer then designs and implements grammars and abstract syntax tree (AST) models for the TestSpec DSL and Driver reference design DSL. After the DSLs are debugged, the Test Engineer compiles and debugs the translation function.

If the translation function cannot perform 100% of the TestSpec→Driver translation activity, the Test Engineer designs, implements, or reuses (1) a Spec→UUT interface reformulation mapping, (2) transformation rules, (3) rule application control function(s), and (4) a main program. Additional details of these artifacts and activities will be presented in Section 3. In the last phase, the Test Engineer tests the TestSpec→Driver translator, demonstrates, and deploys it to Test Developers in the application domain.

3. Prototype Siddhartha implementation

We developed a prototype implementation of Siddhartha's conceptual model via Reasoning SDK[1] [11]. Reasoning SDK is a commercial, off-the-shelf (COTS) programming environment for developing program analysis tools and program synthesis tools. Reasoning SDK provides REFINE, DIALECT, and predefined models for Ada, C, FORTRAN, and COBOL. REFINE is an executable, wide-spectrum formal specification language. DIALECT is an LALR(1) grammar definition language. Together REFINE and DIALECT support development of domain-specific languages.

Many of the example code fragments presented in this section are drawn from the implementation of Siddhartha-REQT(SCR log, Ada), a domain-specific TestSpec→Driver translator we developed supporting requirements testing of Ada procedures against SCR models.

Figure 2 represents the prototype's library organization. Nodes represent collections of source code. Each node linked to nodes below it imports the services provided by the linked nodes.

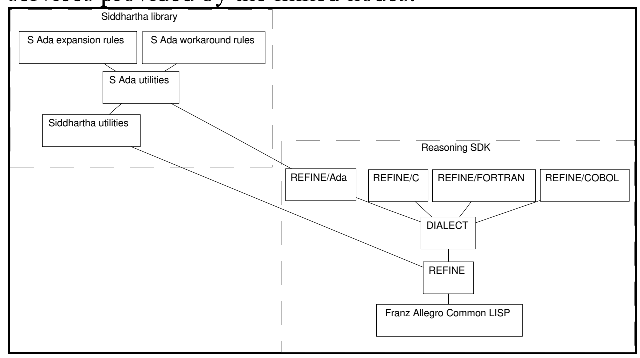

Figure 2: Siddhartha library organization

The structure of Reasoning SDK is shown on the lower right. The Siddhartha library is shown on the upper left. Siddhartha utilities are programming language-independent and are used to test and analyze TestSpec→Driver translators under development. The Siddhartha library currently contains a collection of utilities and transformation rules supporting Ada.

We developed a reference architecture [8] to which TestSpec→Driver translators must conform. The reference architecture for TestSpec→Driver translators is domain-independent and distinct from domain-specific reference designs for Drivers. Figure 3 depicts the TestSpec→Driver translator reference architecture.

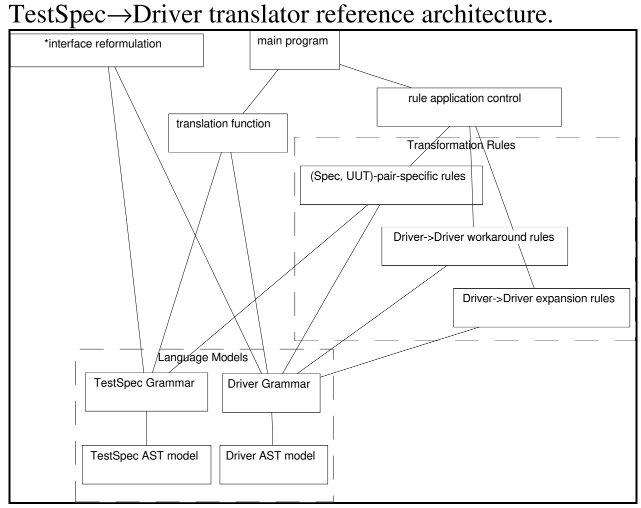

Figure 3: TestSpec→Driver translator reference architecture

The translator reference design is comprised of language models, a translation function, transformation rules, rule application control function(s), a Spec→UUT interface reformulation mapping, and a main program. Language models represent the TestSpec DSL and

---

1 ™ Reasoning SDK, Software Refinery, REFINE, DIALECT, REFINE/Ada, REFINE/C, REFINE/FORTRAN, and REFINE/COBOL are trademarks of Reasoning Systems Incorporated, Mountain View, CA, U.S.A..

Driver reference design DSL. A language model is comprised of an AST model and a grammar. Because Reasoning SDK has predefined models for Ada, C, FORTRAN, and COBOL, an explicit AST model and grammar may not be needed if the TestSpec DSL or Driver reference design DSL is expressed in Ada, C, FORTRAN, or COBOL. We will discuss more about this later.

The translation function maps TestSpecs to Drivers in the reference design. The structure of translation functions is presented in Figure 4. This function maps Boolean REFINE expressions into corresponding Ada expressions as required in a Driver reference design.

```
function TR (X: RE::V-PROGRAM-CONSTRUCT): USER-OBJECT
  = let (RESULT = MAKE-OBJECT('USER-OBJECT))
      RE::V-PROGRAM-CONSTRUCT(X)
      --> (X = `@L & @R'
           --> RESULT = `##r ADA (@(TR (L)) and @(TR (R)))#')
            & (X = `@L or @R'
               --> RESULT = `##r ADA (@(TR (L)) or @(TR (R)))#')
            & (X = `~@P' --> RESULT = `##r ADA (not (@(TR (P))))#')
            & (X = `@L = @R'
               --> RESULT = `##r ADA (@(TR (L)) = @(TR (R)))#')
            & (X = `@L + @R'
               --> RESULT = `##r ADA (@(TR (L)) + @(TR (R)))#')
            & (X = `@L - @R'
               --> RESULT = `##r ADA (@(TR (L)) - @(TR (R)))#')
            & (X = `@L * @R'
               --> RESULT = `##r ADA (@(TR (L)) * @(TR (R)))#')
            & (X = `@L / @R'
               --> RESULT = `##r ADA (@(TR (L)) / @(TR (R)))#')
            & (X = `true' --> RESULT = `##r ADA TRUE#')
            & (X = `false' --> RESULT = `##r ADA FALSE#')
            & (RE::LITERAL-INTEGER-OP(X) & RE::L-INTEGER(X) = N
               --> RESULT
                     = SET-ATTRS
                         (MAKE-OBJECT('INTEGER-LITERAL),
                          'INTEGER-LITERAL-VALUE, N))
            & (RE::LITERAL-REAL-OP(X) & RE::L-REAL(X) = N
               --> RESULT = REAL-TO-REAL-LITERAL(N))
            & (RE::BINDING-REF(X) & RE::BINDINGNAME(X) = SYM
               --> RESULT
                     = SET-ATTRS
                         (MAKE-OBJECT('SIMPLE-NAME), 'SIMPLE-NAME-SYMBOL,
                          SYM));
      RESULT
```

Figure 4: Example recursive translation function

A Spec→UUT interface reformulation mapping is a context-free, finite map from identifiers appearing in TestSpecs to corresponding expressions and statements appearing in Drivers. An example map is presented in Figure 5. This maps identifiers appearing in logs generated by the SCR* Toolset Simulator [5] of an SCR [6] model of a digital, research flight control system (RFCS) into corresponding expressions in the RFCS's Ada library.

```
var *PSFCC-BUILD-5-ID-MAP*: IDENTIFIER-MAPPING
  = {| 'AFU -> MY-COPY-TERM
                 (MY-REF-TO
                    (`##r ADA DPR_INTERFACE.EPDFAIL.AFU', *PSFCC-BUILD-5*)),
...
         , 'EPDFAIL -> MY-COPY-TERM(`##r ADA EPDFAIL'),
        'INITIALIZATION_DONE
          -> MY-COPY-TERM(`##r ADA INITIALIZATION_DONE'),
        'INITIA_CNT -> MY-COPY-TERM(`##r ADA INITIA_CNT'),
        'STATE_COUNT -> MY-COPY-TERM(`##r ADA STATE_COUNT'),
        'ENGAGE_OUT_POSITION
          -> MY-COPY-TERM(`##r ADA ENGAGE_OUT_POSITION')
       |}
```

Figure 5: Example Spec→UUT interface reformulation mapping (fragment)

Transformation rules handle context-sensitive and unrestricted aspects of TestSpec→Driver translation. Driver→Driver expansion add syntactic details (needed for correct compilation) to a kernel Driver generated by the translation function. Figure 6 presents an example Driver→Driver expansion transformation rule. This rule adds a "with" clause to an Ada compilation unit when needed.

```
rule ADD-WITH-CLAUSE-UNIT (X: COMPILATION-UNIT)
  X = `##r ADA #compilation-unit #context-clause $CONTEXT-CLAUSES
       #unit-content @DECL-ITEM#'
    & REF in DESCENDANTS-OF-CLASS(DECL-ITEM, 'IDENTIFIER-REF)
    & ID = REF-TO(REF) & DEFINED?(ID)
    & UNIT-ID = LIBRARY-UNIT-ID-OF-ID(ID) & DEFINED?(UNIT-ID)
    & ~TERM-EQUAL?(UNIT-ID, `##r ADA #identifier STANDARD#')
    & fa (C)
        (C in CONTEXT-CLAUSES
           => ~(ex (WC, WITH-ID)
                  (WC in DESCENDANTS-OF-CLASS(C, 'WITH-CLAUSE)
                     & WC = `##r ADA with @WITH-ID;#'
                     & REF-TO(WITH-ID) = UNIT-ID)))
  --> X = `##r ADA #compilation-unit #context-clause
            $CONTEXT-CLAUSES with @NEW-UNIT-ID; #unit-content @DECL-ITEM#'
        & NEW-UNIT-ID = MAKE-IDENTIFIER-REF(UNIT-ID, true)
```

Figure 6: Example Driver→Driver expansion rule

Driver→Driver workaround transformation rules correct or "work around" faults introduced by earlier Driver generation activities. For example the SCR* Toolset Simulator has a fault that can cause log state identifiers to be repeated. We developed a translation function for Siddhartha-REQT(SCR log, Ada) that maps log states to Ada block statements. This simulator fault engenders Ada block statements with repeated identifiers, which is not legal Ada. Figure 7 presents the Driver→Driver workaround transformation rule that corrects Drivers with repeated block statement identifiers.

```
rule RENAME-BLOCK-STATEMENT (X: STATEMENT-SEQUENCE)
  STATEMENT-SEQUENCE-STATEMENTS(X)
    = [$BEFORE, FRST, $MIDDLE, SCND, $AFTER]
    & FRST = `@BLOCK-ID:
                declare STATE: constant STRING := @BLOCK-ID-STRING;
               begin @A1
                    exception #exception-handler $H1 end @BLOCK-ID-REF;#'
    & SCND = `@BLOCK-ID-2:
                declare STATE: constant STRING := @BLOCK-ID-STRING-2;
               begin @A2
                    exception #exception-handler $H2 end @BLOCK-ID-REF-2;#'
    & TERM-EQUAL?(BLOCK-ID, BLOCK-ID-2)
    & TERM-EQUAL?(BLOCK-ID-STRING, BLOCK-ID-STRING-2)
    & TERM-EQUAL?(BLOCK-ID-REF, BLOCK-ID-REF-2)
  --> STATEMENT-SEQUENCE-STATEMENTS(X)
        = [$BEFORE, FRST, $MIDDLE, SCND, $AFTER]
        & FRST = `@BLOCK-ID:
                    declare STATE: constant STRING := @BLOCK-ID-STRING;
                   begin @A1
                               exception #exception-handler $H1 end @BLOCK-ID-REF;#'
        & SCND = `@BLOCK-ID-NEW:
                    declare STATE: constant STRING := @BLOCK-ID-STRING-NEW;
                   begin @A2
                               exception #exception-handler $H2 end @BLOCK-ID-REF-NEW;#'
        & BLOCK-ID-NEW
            = SET-ATTRS
                (COPY-TERM(BLOCK-ID),
                               'IDENTIFIER-NAME,
                 STRING-TO-SYMBOL
                                ([$symbol-to-string(IDENTIFIER-NAME(BLOCK-ID)),$"_NEXT"],
                               "sid"))
        & BLOCK-ID-STRING-NEW
            = SET-ATTRS
                (make-object('string-literal),
                               'STRING-LITERAL-VALUE,
                 symbol-to-string(identifier-name(block-id-new)))
        & BLOCK-ID-REF-NEW = MAKE-IDENTIFIER-REF(BLOCK-ID-NEW, true)
```

Figure 7: Example Driver→Driver workaround rule

(Spec, UUT) pair-specific transformation rules augment the context-free Spec→UUT interface reformulation mapping. Figure 8 presents an example (Spec, UUT) pair-specific transformation rule. This rule maps an equation featuring the RFCS SCR model variable EPDFAIL. In the RFCS SCR model EPDFAIL is of Boolean type, but in the RFCS Ada library, EPDFAIL is a bit vector. This translation is context-sensitive (i.e., depends on the value of EPDFAIL) and cannot be represented by the context-free Spec→UUT interface reformulation mapping.

```
rule PSFCC-FIX-EPDFAIL-EQUALITY (X: EQUALITY)
  X = `##r ADA (EPDFAIL = @BLAH)#'
    & DPR-INTERFACE-EPDFAIL
        = MY-COPY-TERM
            (MY-REF-TO(`##r ADA #expanded-name
                                 DPR_INTERFACE.EPDFAIL#', *PSFCC-BUILD-5*))
  --> (SIMPLE-NAME-SYMBOL(BLAH) = 'FALSE
       --> X = `##r ADA
                  (@DPR-INTERFACE-EPDFAIL
                    = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                       0))#')
        & (SIMPLE-NAME-SYMBOL(BLAH) = 'TRUE
           --> X = `##r ADA
                      (@DPR-INTERFACE-EPDFAIL
                        = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                           1))#')
```

Figure 8: Example (Spec, UUT) pair-specific transformation rule

As their name implies, rule application control functions apply transformation rules. Figure 9 presents an example rule application control function. These functions use the built-in REFINE function POSTORDER-TRANSFORM, which traverses the AST provided as first argument in postorder (i.e., bottom-up). Named rules are tested in the order listed. The last argument, true, indicates that rule application is to be performed exhaustively (i.e., rule application restarts at the root of the AST after each successful rule application).

```
function NEW-EXPAND (X: COMPILATION-UNIT): COMPILATION-UNIT
  = POSTORDER-TRANSFORM
      (X, ['ATOMIC-NAME-TO-EXPANDED-NAME,
           'ADD-TYPE-CONVERSION-TO-EQUALITY-RHS,
           'PSFCC-FIX-EPDFAIL-EQUALITY,
           'PSFCC-FIX-ENGAGE-OUT-POSITION-EQUALITY,
           'PSFCC-DELETE-INVISIBLE-VARIABLES,
           'RENAME-BLOCK-STATEMENT,
          'ADD-WITH-CLAUSE-UNIT,
           'ADD-USE-CLAUSE],
       true)
```

Figure 9: Example rule application control function

The main program sequentially parses the TestSpec file into an AST, applies the translation function to the TestSpec AST, applies the rule application control function to the generated kernel Driver AST, and prints the final Driver AST to a source code file.

4. Example TestSpec→Driver translators

This section briefly describes two domain-specific TestSpec→Driver translators we developed. Siddhartha-REQT(SCR log, Ada) is the name of the translator that maps logs generated by the SCR* Toolset Simulator into Drivers expressed as Ada procedures. Siddhartha-REGR(Ada, Ada) is the name of the translator that first synthesizes regression TestSpecs, expressed as Ada procedure declarations, and maps them into Drivers expressed as Ada procedure bodies. Both translators generate Drivers expressed in reference designs that accommodate Ada procedure UUTs with insufficiently defined interfaces.

4.1 Siddhartha-REQT(SCR log, Ada)

Figure 10 presents Siddhartha-REQT(SCR log, Ada)'s domain model diagram:

- UUTs are Ada main procedures,
- Specs are SCR models,
- TestSpecs are logs generated by the SCR* Toolset Simulator,
- Drivers are Ada procedure bodies,
- Interface Mappings are REFINE global variables of mapping type (of the kind shown in Figure 5).
- The targeted test data selection criterion (TDSC) is to cover every SCR model event table cell with at least one test (although this TDSC has no influence on development of the TestSpec→Driver translator).

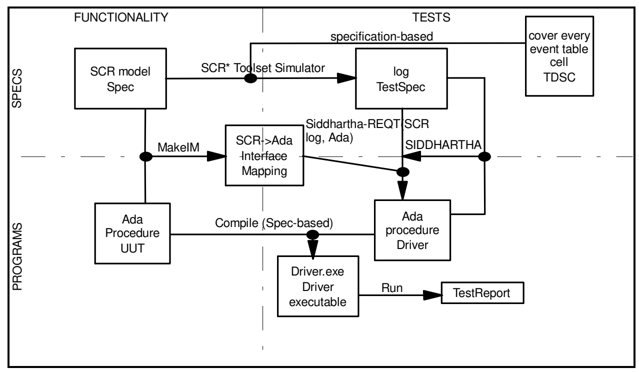

Figure 10: Siddhartha-REQT(SCR log, Ada)'s domain model diagram

We wanted Drivers to be expressed in Ada so they can programmatically control and observe Ada UUTs. Reasoning SDK provides a predefined Ada model, REFINE/Ada. We faced a choice whether to represent the Driver reference design explicitly via a new DIALECT grammar or whether to represent it implicitly within the translation function (thereby reusing the REFINE/Ada model). The REFINE/Ada model resisted all attempts to reuse parts of it in a new grammar for the Driver reference design. Thus we chose to implicitly represent the Driver reference design DSL within the translation function.

Figure 11 presents an example log generated by the SCR* Toolset Simulator. Logs such as this are TestSpecs for Siddhartha-REQT(SCR log, Ada).

Figure 12 presents an example Driver generated by Siddhartha-REQT(SCR log, Ada) from a log file such as the one shown above. Careful inspection reveals their corresponding control and data structures.

```
--- Start State  -------------------------------------
Aileron_Cmd = 0.                        AFU = 0.
CHAN1_3 = 0.                            APCVCS = 0.
...
--- Start State  -------------------------------------
Aileron_Cmd = 0.                        AFU = 0.
CHAN1_3 = 0.                            APCVCS = 0.
...
--- State Edit Begin  ----------------------
Edited new value of Engage_Out = None
Edited new value of PSFCC_State = Parameter_Mod
Edited new value of Disengage_Request = 1
--- State Edit Complete  ----------------------
--- State 1 ---------------------------------------------
Disengage_Request = 0
--- State 2 ---------------------------------------------
Table_Num_MC = 1.0                      PSFCC_State = Armed
                                        ARM_ENG_FAIL = 0
                                        RARML = 1
```

Figure 11: Example requirements TestSpec (fragment)

Figure 13 shows a fragment from the translation function of Siddhartha-REQT(SCR log, Ada). This "translation production" maps a log fragment such as State 1 or State 2 in Figure 11 to an Ada block statement. The first equation in such a log state represents the SCR model input variable taking on a new value. This translates into an assignment statement for the Ada library global variable it represents. After this assignment statement the UUT is invoked. Subsequent equations in such a log state indicate that intermediate and output SCR model variables changed value in response to the input event assignment.

Transformation rules next operate on the generated Driver kernel. We presented sample transformation rules used by Siddhartha-REQT(SCR log, Ada) in Figures 6, 7, and 8. We presented its rule application control function in Figure 9.

4.2 Siddhartha-REGR(Ada, Ada)

Figure 14 presents Siddhartha-REGR(Ada, Ada)'s domain model diagram. In this application domain

- UUTs are Ada procedures,
- Specs are earlier version counterparts of UUTs, assumed functionally equivalent,
- TestSpecs are Ada procedure declarations generated by a data flow analysis tool developed by the Test Engineer,
- Drivers are Ada procedure bodies generated by Siddhartha-REGR(Ada, Ada), and
- Driver-Wrappers are manually-coded Ada procedure bodies that invoke Drivers with appropriate input data determined by the TDSC of covering all UUT branches.

```
with DPR_INTERFACE; use DPR_INTERFACE;
with DATA_TYPES; use DATA_TYPES;
with PSFCC_MAIN; use PSFCC_MAIN;
with ENGAGE_LOGIC; use ENGAGE_LOGIC;
with CL_GLOBAL;
with VARIABLE_RESPONSE;
with TEXT_IO;
procedure FUBAR is
  task SET_E701E_RDY_TASK is end;
  task body SET_E701E_RDY_TASK is begin
    loop delay 1.0; DPR_INTERFACE.E701E_RDY := 1; end loop;
  exception
    when others
      => TEXT_IO.PUT_LINE
           ("an exception occured in SET_E701E_RDY_TASK");
  end SET_E701E_RDY_TASK;
begin
  INITIAL_STATE: declare STATE: constant STRING := "INITIAL";
  begin
    TEXT_IO.PUT_LINE
      ( (((((("In state " & STATE) & ", does ")
              & "DPR_INTERFACE.RAILCD")
             & " = ")
            & "2#0.0#e1")
           & "? ")
          & BOOLEAN ' IMAGE
              ((DPR_INTERFACE.RAILCD = DATA_TYPES.REAL ( 2#0.0#e1)))));
    TEXT_IO.PUT_LINE
      ( (((((("In state " & STATE) & ", does ")
              & "DPR_INTERFACE.EPDFAIL.AFU")
             & " = ")
            & "2#0.0#e1")
           & "? ")
          & BOOLEAN ' IMAGE
              ((DPR_INTERFACE.EPDFAIL.AFU
                 = DATA_TYPES.INTEGER_BIT ( 2#0.0#e1)))));
...
  PSFCC_MAIN.PSFCC_ADA_CLAW;
  exception
    when others
      => TEXT_IO.PUT_LINE ( "Exception raised in INITIAL state.");
  end INITIAL_STATE;
…
  STATE_1:
    declare STATE: constant STRING := POSITIVE ' IMAGE (1);
  begin CL_GLOBAL.CHAN1_3 := 1; PSFCC_MAIN.PSFCC_ADA_CLAW;
  exception
    when others
      => TEXT_IO.PUT_LINE
           ( (("Exception raised in state " & STATE) & "."));
  end STATE_1;
  STATE_2:
    declare STATE: constant STRING := POSITIVE ' IMAGE (2);
  begin CL_GLOBAL.CHAN1_4 := 1; PSFCC_MAIN.PSFCC_ADA_CLAW;
  exception
    when others
      => TEXT_IO.PUT_LINE
           ( (("Exception raised in state " & STATE) & "."));
  end STATE_2;
  STATE_3:
    declare STATE: constant STRING := POSITIVE ' IMAGE (3);
  begin
    VARIABLE_RESPONSE.VRR_REC.CNTRL1 := 1;
    PSFCC_MAIN.PSFCC_ADA_CLAW;
  exception
    when others
      => TEXT_IO.PUT_LINE
           ( (("Exception raised in state " & STATE) & "."));
  end STATE_3;
  STATE_4:
    declare STATE: constant STRING := POSITIVE ' IMAGE (4);
  begin
    DPR_INTERFACE.DISENGAGE_REQUEST := 1;
    PSFCC_MAIN.PSFCC_ADA_CLAW;
    TEXT_IO.PUT_LINE
      ( (((((("In state " & STATE) & ", does ")
              & "PSFCC_MAIN.PSFCC_STATE")
             & " = ")
            & "DISENGAGED")
           & "? ")
          & BOOLEAN ' IMAGE ((PSFCC_MAIN.PSFCC_STATE = DISENGAGED))));
    TEXT_IO.PUT_LINE
      ( (((((("In state " & STATE) & ", does ")
              & "ENGAGE_LOGIC.ENGAGE_OUT")
             & " = ")
            & "TABLE_NUMBER")
           & "? ")
          & BOOLEAN ' IMAGE ((ENGAGE_LOGIC.ENGAGE_OUT = TABLE_NUMBER)))
       );
    TEXT_IO.PUT_LINE
      ( (((((("In state " & STATE) & ", does ")
              & "ENGAGE_OUT_POSITION")
             & " = ")
            & "8")
           & "? ")
          & BOOLEAN ' IMAGE
              ((ENGAGE_LOGIC.ENGAGE_OUT
                 = ENGAGE_LOGIC.TEST_LIST_INDEX_TYPE
                      ( ENGAGE_LOGIC.TEST_LIST_INDEX_TYPE ' VAL
                          ( POSITIVE ( 8)))))));
    TEXT_IO.PUT_LINE
      ( (((((("In state " & STATE) & ", does ")
              & "DPR_INTERFACE.PSFCC_STATE_MODE_ID.ARM_ENG_FAIL")
             & " = ")
            & "8")
           & "? ")
          & BOOLEAN ' IMAGE
              ((DPR_INTERFACE.PSFCC_STATE_MODE_ID.ARM_ENG_FAIL
                 = DATA_TYPES.INTEGER ( 8)))));
  exception
    when others
      => TEXT_IO.PUT_LINE
           ( (("Exception raised in state " & STATE) & "."));
  end STATE_4;
  abort SET_E701E_RDY_TASK;
  TEXT_IO.PUT_LINE ( "Test completed.");
exception
  when others
    => TEXT_IO.PUT_LINE
         ( "An exception was raised during elaboration.");
end FUBAR;
```

Figure 12: Example generated regression Driver

```
& (X = `##r STSL
          --- state @N
            -----------------------------------------------
            @ID-REF = @EXPR $EQUATIONS'
    --> RESULT
          = `##r ADA
              @BLOCK-ID:
                declare
                  STATE:
                    constant STRING
                      := POSITIVE ' IMAGE[@N-INT-LIT];
              begin
                @(ID-MAP (STSL-SIMPLE-NAME-SYMBOL(ID-REF))) :=
                  @(H (EXPR, UUT, LIB, ID-MAP));
                @(MY-COPY-TERM (UUT-ID-REF))[];
                $image
                   (lambda (E) H(E, UUT, LIB, ID-MAP), EQUATIONS)
                exception
                  when others
                    => @(MY-COPY-TERM (TEXT-IO-PUT-LINE))
                         [ (("Exception raised in state "
                             & STATE)
                            & ".")];
              end @BLOCK-ID-REF;#'
         & BLOCK-ID
             = APPEND-STRING-TO-NAMED-IDENTIFIER
                 (`##r ADA #identifier STATE#',
                  concat("_", FORMAT(false, "~/pp/", N)))
         & BLOCK-ID-REF = MAKE-IDENTIFIER-REF(BLOCK-ID, true)
         & N-INT-LIT
             = SET-ATTRS
                 (MAKE-OBJECT('INTEGER-LITERAL),
                  'INTEGER-LITERAL-VALUE, N))
```

Figure 13: Siddhartha-REQT(SCR log, Ada)'s translation function (fragment)

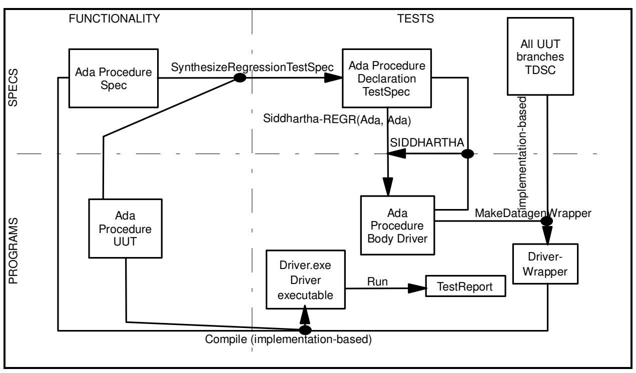

Figure 14: Siddhartha-REGR(Ada, Ada)'s domain model diagram

```
procedure OL_FOUR_CHAN_CALC is F86: DATA_TYPES_UUT.REAL;
begin
  CL_P_UUT.PV8 := (APK11 * ALIMIT ( APK9, APK8, AV11));
  F86 := (((abs PHIEST) - 15.0) * 2828.0e-5);
  CL_P_UUT.PV9 :=
    ((CL_Y_UUT.YV8 * APK10) * ALIMIT ( 0.0, 0.9129, F86));
  CL_R_UUT.RV8 := ALIMIT ( (- APK13), APK13, (APK12 * AV4));
end OL_FOUR_CHAN_CALC;
```

Figure 15: Example regression UUT

Figure 15 presents an example regression UUT. Its Spec is almost identical. Notice the procedure takes no formal parameters.

Figure 16 presents the TestSpec generated by the data flow analysis tool developed by the Test Engineer. The tool discovers approximate, effective interfaces for Ada procedures, taking global variables into account. Matched global variables appearing in the approximate, effective interfaces of UUT and Spec are represented by formal parameters in the TestSpec. This immediately reveals any interface mismatches between UUT and Spec and provides a functional abstraction around both UUT and Spec.

```
with DATA_TYPES_UUT; with DATA_TYPES_SPC;
procedure OL_FOUR_CHAN_CALC_REGRESSION_TEST
  ( PHIEST_IMG: in DATA_TYPES_UUT.REAL := 0.0;
    YV8_IMG: in DATA_TYPES_UUT.REAL := 0.0;
    APK11_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    AV11_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    APK8_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    APK9_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    APK10_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    APK12_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    AV4_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    APK13_IMH: in DATA_TYPES_UUT.REAL := 0.0;
    PV8_OMG_UUT: out DATA_TYPES_UUT.REAL;
    PV9_OMG_UUT: out DATA_TYPES_UUT.REAL;
    RV8_OMG_UUT: out DATA_TYPES_UUT.REAL;
    PV8_OMG_SPC: out DATA_TYPES_SPC.REAL;
    PV9_OMG_SPC: out DATA_TYPES_SPC.REAL;
    RV8_OMG_SPC: out DATA_TYPES_SPC.REAL;
    TEST_PASSED: out BOOLEAN);
```

Figure 16: Example generated regression TestSpec

```
with DATA_TYPES_UUT; use DATA_TYPES_UUT;
with TEXT_IO;
with CL_GLOBAL_UUT;
with CL_GLOBAL_SPC;
with CL_Y_UUT;
with CL_Y_SPC;
with OLCLAW_UUT;
with OLCLAW_SPC;
with CL_P_UUT;
with CL_R_UUT;
with CL_P_SPC;
with CL_R_SPC;
with DATA_TYPES_SPC;
procedure OL_FOUR_CHAN_CALC_REGRESSION_TEST
  ( PHIEST_IMG: in DATA_TYPES_UUT.REAL := 0.0;
    YV8_IMG: in DATA_TYPES_UUT.REAL := 0.0;
    PV8_OMG_UUT: out DATA_TYPES_UUT.REAL;
    PV9_OMG_UUT: out DATA_TYPES_UUT.REAL;
    RV8_OMG_UUT: out DATA_TYPES_UUT.REAL;
    PV8_OMG_SPC: out DATA_TYPES_SPC.REAL;
    PV9_OMG_SPC: out DATA_TYPES_SPC.REAL;
    RV8_OMG_SPC: out DATA_TYPES_SPC.REAL;
    TEST_PASSED: out BOOLEAN)
  is begin
  CL_GLOBAL_UUT.PHIEST := DATA_TYPES_UUT.REAL ( PHIEST_IMG);
  CL_GLOBAL_SPC.PHIEST := DATA_TYPES_SPC.REAL ( PHIEST_IMG);
  CL_Y_UUT.YV8 := DATA_TYPES_UUT.REAL ( YV8_IMG);
  CL_Y_SPC.YV8 := DATA_TYPES_SPC.REAL ( YV8_IMG);
  OLCLAW_UUT.OL_FOUR_CHAN_CALC;
  OLCLAW_SPC.OL_FOUR_CHAN_CALC;
  TEST_PASSED :=
    (((TRUE = TRUE)) and
      ((((CL_P_UUT.PV8 = CL_P_SPC.PV8) and
          (CL_P_UUT.PV9 = CL_P_SPC.PV9))
        and (CL_R_UUT.RV8 = CL_R_SPC.RV8))));
  PV8_OMG_UUT := CL_P_UUT.PV8;
  PV9_OMG_UUT := CL_P_UUT.PV9;
  RV8_OMG_UUT := CL_R_UUT.RV8;
  PV8_OMG_SPC := CL_P_SPC.PV8;
  PV9_OMG_SPC := CL_P_SPC.PV9;
  RV8_OMG_SPC := CL_R_SPC.RV8;
exception
  when others => TEXT_IO.PUT_LINE ( "Exception Raised.");
end OL_FOUR_CHAN_CALC_REGRESSION_TEST;
```

Figure 17: Example generated regression Driver

Figure 17 presents the Driver generated from the TestSpec above by Siddhartha-REGR(Ada, Ada).

Figure 18 presents the translation function for Siddhartha-REGR(Ada, Ada). This translation function is very different from the translation function for Siddhartha-REQT(SCR log, Ada). This translation function is not recursive and relies on a large number of highly-specialized translation functions. This translation function must cross a much larger syntactic gap than Siddhartha-REQT(SCR log, Ada)'s translation function. TestSpecs for Siddhartha-REGR(Ada, Ada) are declarative, while Drivers are fully operational. As in Siddhartha-REQT(SCR log, Ada), Siddhartha-REGR(Ada, Ada) encodes the Driver reference design within the translation function.

5. Evaluation

We evaluated these two domain-specific TestSpec→Driver translators with respect to a subset of tests performed by the NASA Software IV&V Facility on the Production Support Flight Control Computer (PSFCC) [2] RFCS. Each selected NASA test was replicated twice: first by generated Drivers and second by manually-coded Drivers that are arguably equivalent to Drivers generated by NASA's custom unit test support tool.

We designed the factorial experiment shown in Table 1. We were interested in seeing interactions between Siddhartha vs. Manual, requirements tests vs. regression tests, whether NASA found a fault by their test or not, and whether the TestSpec was small or large.

Figure 19 clearly shows the interaction between (Siddhartha, Manual) and (Regression, Requirements) for total Driver development duration (TotalDev). Each data point is the average of four experiments. Siddhartha provided no reduction in total development time for requirements Drivers because in the Siddhartha cases the Test Developer manually operated the SCR* Toolset Simulator, while in the Manual cases the Test Developer engaged in roughly the same cognitive activity by coding a Driver directly from an SCR event table cell. Siddhartha provided significant reduction in total development time for regression drivers because the data flow analyzer discovered approximate, effective UUT and Spec interfaces much faster than manual discovery.

Figure 20 clearly shows the interaction between (Siddhartha, Manual) and (Regression, Requirements) for failures detected and exceptions raised by developed Drivers. Each data point is the average of four experiments. Generated regression Drivers found every failure and exception found by manual-coded regression drivers because the data flow analyzer produced more accurate TestSpecs than manual analysis. Generated requirements Drivers found every failure found by manually-coded requirements Drivers plus significantly more failures because logs generated by the SCR* Toolset Simulator contain considerably more test-relevant information than SCR event table cells alone.

```
function SYNTHESIZE-REGRESSION-TEST-DRIVER-ORACLE
  (TS: COMPILATION-UNIT): COMPILATION-UNIT
  = TS = `##r ADA
           $CONTEXT-CLAUSES
             procedure @TS-ID
               ( $PARAMETER-SPECIFICATIONS; TEST_PASSED: out BOOLEAN);'
      & UUT-ID-REF
          = MAKE-IDENTIFIER-REF
              (GET-IDENTIFIER-ORIGINAL
                 (SUBPROGRAM-SPECIFICATION-IDENTIFIER
                    (SUBPROGRAM-BODY-SPECIFICATION
                       (COMPILATION-UNIT-CALLS-UUT(TS)))),
               true)
      & SPC-ID-REF
          = MAKE-IDENTIFIER-REF
              (GET-IDENTIFIER-ORIGINAL
                 (SUBPROGRAM-SPECIFICATION-IDENTIFIER
                    (SUBPROGRAM-BODY-SPECIFICATION
                       (COMPILATION-UNIT-CALLS-SPC(TS)))),
               true)
    --> TS = `##r ADA
               $CONTEXT-CLAUSES
                 procedure @TS-ID
                   ( $IMF-PARAMETER-SPECIFICATIONS;
...
                   TEST_PASSED: out BOOLEAN)
                   is $OMF-UUT-TEMP-DECLARATIONS
                      $OMF-SPC-TEMP-DECLARATIONS
                      $OUF-UUT-TEMP-DECLARATIONS
                      $OUF-SPC-TEMP-DECLARATIONS
                      $BMF-IN-TEMP-DECLARATIONS
                      $BUF-IN-UUT-TEMP-DECLARATIONS
                      $BUF-IN-SPC-TEMP-DECLARATIONS
                   begin
                   $IMG-ASSIGNMENT-STATEMENTS
                   $IUG-UUT-ASSIGNMENT-STATEMENTS
                   $IUG-SPC-ASSIGNMENT-STATEMENTS
                   @UUT-ID-REF
                     ( $IMF-UUT-PARAMETER-ASSOCIATIONS,
                       $IUF-UUT-PARAMETER-ASSOCIATIONS,
                       $BMF-UUT-PARAMETER-ASSOCIATIONS);
                   @SPC-ID-REF
                     ( $IMF-SPC-PARAMETER-ASSOCIATIONS,
                       $IUF-SPC-PARAMETER-ASSOCIATIONS,
                       $BMF-SPC-PARAMETER-ASSOCIATIONS);
                   TEST_PASSED :=
                     ((@OMF-TEMP-EQUALITIES) and (@OMG-EQUALITIES));
                   $OMG-UUT-ASSIGNMENT-STATEMENTS
                   $OMG-SPC-ASSIGNMENT-STATEMENTS
                   exception
                     when others
                       => @TEXT-IO-PUT-LINE ( "Exception Raised.");
                 end @TS-ID-REF;'
        & TS-ID-REF = MAKE-IDENTIFIER-REF(TS-ID, true)
        & IMF-PARAMETER-SPECIFICATIONS
            = [PS | (PS: PARAMETER-SPECIFICATION)
                 PS in PARAMETER-SPECIFICATIONS
                   & NAMED-IDENTIFIER-ENDS-IN?
                       (SINGLETON-SEQ-ELEMENT
                          (PARAMETER-SPECIFICATION-IDENTIFIERS(PS)),
                        "_IMF")]
...
       & IMG-ASSIGNMENT-STATEMENTS
            = MAKE-IN-MATCHED-GLOBAL-ASSIGNMENT-STATEMENTS
                (IMG-PARAMETER-SPECIFICATIONS)
...
       & TEXT-IO-PUT-LINE
            = MY-COPY-TERM
                (MY-REF-TO
                   (`##r ADA TEXT_IO.PUT_LINE', *PSFCC-BUILD-5-UUT*));
    TS
```

Figure 18: Siddhartha-REGR(Ada, Ada) translation function (fragment)

Table 1: Factorial experiment design

Reyes ID	Tst Type	IV&V Fault?	TS Size	Method	UUT ID	NASA Test ID	table ID/output vars tested	row	col
1	reqt.	yes	big	manual	psfcc_ada_claw	8006	arm_eng_fail	3	3
2	reqt.	yes	big	Siddhartha	psfcc_ada_claw	8006	arm_eng_fail	3	3
3	reqt.	yes	small	manual	psfcc_ada_claw	8016	display_count	1	3
4	reqt.	yes	small	Siddhartha	psfcc_ada_claw	8016	display_count	1	3
5	reqt.	no	big	manual	psfcc_ada_claw	8146	apcvcs	1	3
6	reqt.	no	big	Siddhartha	psfcc_ada_claw	8146	apcvcs	1	3
7	reqt.	no	small	manual	psfcc_ada_claw	8015	display_count	1	2
8	reqt.	no	small	Siddhartha	psfcc_ada_claw	8015	display_count	1	2
9	regress.	yes	big	manual	engage_executive	DF-C-037(IB)	stored_mode_id, engage_out	NA	NA
10	regress.	yes	big	Siddhartha	engage_executive	DF-C-037(IB)	stored_mode_id, engage_out	NA	NA
11	regress.	yes	small	manual	casint	DF-C-010	yout	NA	NA
12	regress.	yes	small	Siddhartha	casint	DF-C-010	yout	NA	NA
13	regress.	no	big	manual	initc1_a		80 vars	NA	NA
14	regress.	no	big	Siddhartha	initc1_a		80 vars	NA	NA
15	regress.	no	small	manual	ol_four_chan_calc	DF-C-004 del AV4L	pv8, pv9, rv8	NA	NA
16	regress.	no	small	Siddhartha	ol_four_chan_calc	DF-C-004 del AV4L	pv8, pv9, rv8	NA	NA

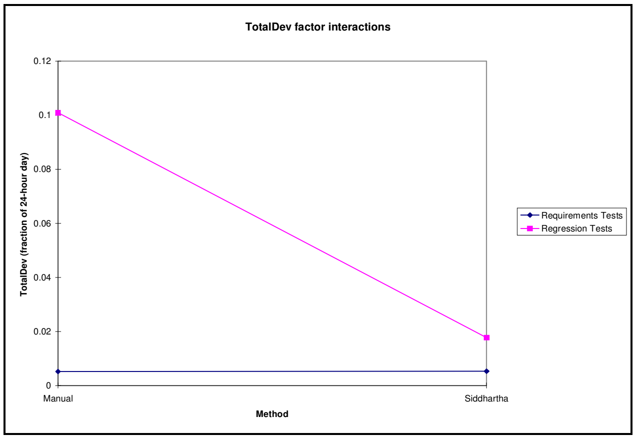

Figure 19: Factor interactions for TotalDev

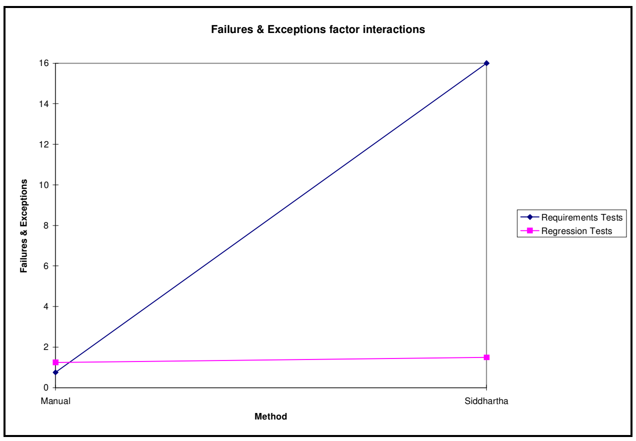

Figure 20: Factor interactions for Failures and Exceptions

6. Conclusions

We presented Siddhartha, a method for developing domain-specific TestSpec→Driver translators. Translators provide essential automation support for SBT in novel application domains. Siddhartha applies the DSL paradigm to solve difficult problems in SBT: discovery and development of DSLs representing TestSpecs and difficult-to-test UUT design styles and development of a language processing tool (i.e., a translator) to provide automation support for the DSLs.

The existence of Siddhartha-REQT(SCR log, Ada) means that insufficient interface definition and nontrivial Spec→UUT interface reformulation are no longer valid excuses for not rigorously testing safety-critical Ada main procedures against SCR models.

The existence of Siddhartha-REGR(Ada, Ada) means that insufficient interface definition is no longer an excuse for not performing rigorous regression unit testing of safety-critical Ada procedures.

7. References

[1] Mark R. Blackburn, Robert D. Busser, Joseph S. Fontaine, Automatic generation of test vectors for SCR-style specifications. *Proceedings of Computer Assurance 1997: Are we Making Progress Toward Computer Assurance?*, IEEE New York, NY, USA.

[2] Carter, J. Production support flight control computers: research capability for F/A-18 aircraft at Dryden Flight Research Center. IN: *16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings*, Irvine, CA, USA, 26-30 Oct. 1997). New York, NY, USA: IEEE, 1997. p. 7.2-8-7.2-23 vol.2.

[3] Roong-Ko Doong; Frankl, P.G. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, April 1994, vol.3, (no.2):101-30.

[4] Gannon, J.; McMullin, P.; Hamlet, R. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, July 1981, vol.3, (no.3):211-23.

[5] Heitmeyer, C.; Bull, A.; Gasarch, C.; Labaw, B. SCR: a toolset for specifying and analyzing requirements. IN: *COMPASS '95. Proceedings of the Tenth Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, 25-29 June 1995). New York, NY, USA: IEEE, 1995. p. 109-22.

[6] Heninger, K.L. Specifying software requirements for complex systems: new techniques and their application. *IEEE Transactions on Software Engineering*, Jan. 1980, vol.SE-6, (no.1):2-13.

[7] Hughes, M.; Stotts, D. Daistish: systematic algebraic testing for OO programs in the presence of side-effects. (1996 International Symposium on Software Testing and Analysis (ISSTA), San Diego, CA, USA, 8-10 Jan. 1996). *SIGSOFT Software Engineering Notes*, May 1996, vol.21, (no.3):53-61.

[8] Erik Mettala, Marc H. Graham, *The Domain Specific Software Architecture Program*, Software Engineering Institute Special Report CMU/SEI-92-SR-9, June 1992.

[9] Poston, Robert M. *Automating specification-based software testing* / Robert M. Poston. Los Alamitos, Calif. : IEEE Computer Society Press, c1996.

[10] Prieto-Diaz, Ruben; Arango, Guillermo; *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[11] *REFINE User Guide, DIALECT User Guide, REFINE/Ada User Guide*, Reasoning Systems, Mountain View, CA.

[12] Vieira, Marlon, *DAS-BOOT Internet Home Page* http://www.ics.uci.edu/~mvieira/das-boot.htm

[13] Weber, R.; Thelen, K.; Srivastava, A.; Krueger, J. Automated validation test generation. IN: *AIAA/IEEE Digital Avionics Systems Conference*. 13th DASC Phoenix, AZ, USA, 30 Oct.-3 Nov. 1994). New York, NY, USA: IEEE, 1994. p. 99-104.