Traceability and Evidence of Correctness of EDSL Abstractions

Martin Sulzmann
Hochschule Karlsruhe - Technik und Wirtschaft
martin.sulzmann@hs-karlsruhe.de

Jürgen Nicklisch-Franken
ICS AG
juergen.nicklisch@ics-ag.de

Axel Zechner
ICS AG
axel.zechner@ics-ag.de

Abstract

One of the main advantages of an EDSL (embedded domain-specific language) is that new abstractions can be coded quickly and easily in the EDSL's host language and are automatically transformed to the basic EDSL primitives. In the context of formal software certification, it is paramount that *evidence* for the *correctness* of these abstractions are provided and that the low-level code resulting from the EDSL primitives can be *traced* to some higher-level artifacts, i.e. some concrete programming abstractions, software requirements etc. We have built an EDSL-based tool-chain for implementing and testing mission critical applications which supports measures to guarantee traceability and provides evidence of correctness of EDSL abstractions. We give an overview of our EDSL approach and practical experiences applying them in the industrial context.

Categories and Subject Descriptors D.2.2 [**Software Engineering**]: Design Tools and Techniques—Software libraries; D.2.5 [**Software Engineering**]: Testing and Debugging—Tracing

Keywords EDSL, Abstractions, Traceability, Correctness

1. Introduction

The application domain of our EDSLs are mission critical embedded control and simulation software for reactive systems. The functionality of these systems increasingly relies on software which must meet high safety standards to guarantee their correct functioning. To ensure predictable performance and memory consumption, such systems are often implemented by following the synchronous execution paradigm. That is, given some input values the application shall produce some output values in every cycle. The cycle time is fixed, so that it needs to work within some predictable amount of time.

The EDSLs we describe here have emerged out of a software project in the Aerospace& Defense area. In the following, we have a quick run through the various aspects of our EDSLs.

EDSL Technology Our design decision was to use a rule-based language, in essence guarded actions, where in each cycle every rule is tried sequentially. If the guard of a rule is satisfied the rule's action will be executed.

The rule-based language, referred to as *Rules EDSL*, is embedded in Haskell. We chose Haskell because of familiarity with the language and Haskell's well-known ability to be a good host for embedded languages.

The architecture and compilation of the basic Rules EDSL to simple if-then statements in C is straightforward. We apply a deep embedding where running the program generates an abstract syntax tree (AST) which can then be processed further in several ways (i.e. generate code, generate documentation, check for correctness). We use singleton types together with phantom types to reflect the AST at the level of types. Technically, we follow [1] and represent constructors of our EDSL via values (type class methods). The meaning of programs is then specified by defining the appropriate type class instances. Thus, we obtain a strongly typed EDSL language.

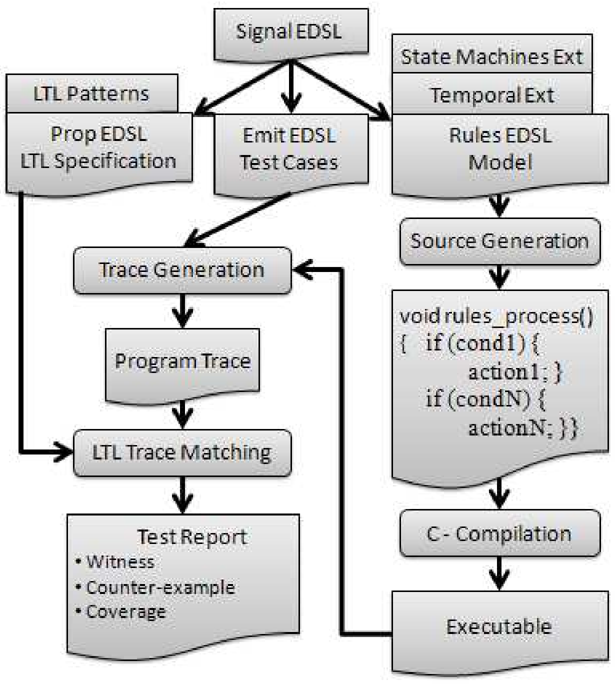
EDSL Abstractions The Rules EDSL offers additional type-safety compared to C but programming in the bare Rules EDSL is still very low level. So far we have built various abstractions using Haskell as our macro language. For standard signal processing we provide operations like detection of rising and falling edges, basic counter operations, temporal primitives such as a `duration`, primitive delays for switch-on and switch-off operations etc. We also offer various kinds of state machines to better capture the semantics of the problem domain. The state machine abstractions range from simple cases to timed state-machines, hierarchical state-machines and state machines with parallel branches.

Traceability For us, the EDSL and its abstraction are only a means to an end. Ultimately, we are only interested in the resulting C code. In our largest industrial application, we find about 2000 primitive rules. Of course, by looking at the resulting C code we hardly can recognize the structure which is present at the higher EDSL level. However, in the context of mission-critical applications which are typically regulated by safety standards, every single line of code needs to be justified and traced back to some higher-level artifact.

The approach pursued by commercial tools such as SCADE is to have a qualified code generator [3]. The upshot of this approach is that the final C code doesn't need to be inspected anymore. It suffices to inspect the software models out of which the C code is generated. For us this is not a viable option. Firstly, formal software qualification [6] is very costly. Secondly, our language model, that is the EDSL abstractions, can change rapidly depending on the problem domain. But this then would require re-qualification.

Our approach is to attach meta data to each rule which explains why this rule was generated. For example, `modeMachine` transitions are translated to rules which maintain the link to the original construct.

Testing and Verification To ensure the correctness of EDSL abstractions we make use of an expressive test and verification system which we have integrated into our framework. At its core we find a *Test EDSL* which consists of a temporal property language to specify the expected outcome of a test and another language to set the inputs of the (synchronous) system under test. Both languages share the sensor signal language with the Rule EDSL. Our interest



void rules_process()
{ if (cond1) {
action1; }
if (condN) {
actionN; }}

Figure 1. EDSL Toolchain Overview

is mainly in run-time verification (i.e. testing) where we execute the system under test by fixing the set of inputs over a certain number of cycles. We obtain a trace log which consists of the input/output values at each cycle. We then perform an off-line analysis of the trace log by matching the trace log against a test property specified as a linear temporal logic (LTL) formula [5].

Thus, we verify the essential building blocks, i.e. the EDSL abstractions. Besides providing strong guarantees about the correctness of the EDSL abstractions, this has the additional benefit that the test properties serve as concise specification of the EDSL abstractions.

But who verifies the verifier, in our case the run-time verification system? Our take on this issue is as follows. The test framework creates detailed reports which allows us to follow each step of the run-time verification system. Technically, we apply a constructive finite trace LTL matching algorithm which provides evidence in form of a proof why a match exists. In doubt, the proof can then be manually inspected. We also collect coverage info to provide feedback if further test cases are required. The formal details are described in [7].

Besides run-time verification we also support model-checking by verifying the C code generated out of the EDSL against the LTL specifications. We make use here of a feature of the SPIN model checker to integrate native C code. For details, we refer to [8].

EDSL Toolchain Overview and Paper Outline Figure 1 summarizes our EDSL toolchain. Section 2 explains our EDSL with a simple example. Sections 3 and 4 demonstrate traceability and testing. Section 5 reports practical experiences and lessons we have learned.

An extended version of this paper with further details (e.g. on testing which we only briefly mention here due to space constraints), the complete set of EDSLs, as well as an example from the Automotive area are freely available via

`http://www.home.hs-karlsruhe.de/~suma0002/REDSL.html`

Related Work As already mentioned, the EDSL technology we employ is by now fairly standard [1]. Our rule-based language is inspired by the Atom DSL [2]. A noteworthy extension is our support for a user-configurable sensor signal type language.

```
durationGtEq condIn limitIn = node "durationGtEq" $ do
  cond <- inpVar "cond" condIn
  limit <- inpVar "limit" limitIn
  res <- outpVar "res" (undefined :: Bool)
  counter <- stateVar "counter" (constE (1::Int))
  mode <- localVar "mode" (undefined :: DurationState)
  mode .=:==.
     modeMachine "durGtEq" Idle
         [ (Idle .---->. InProgress)
               (cond .==. constE True)
         , (InProgress .---->. DurationGtEq)
               ((cond .==. constE True)
               .&&. (valueState counter .==. limit))
         , (InProgress .---->. Idle)
                (cond .==. constE False)
         , (DurationGtEq .---->. Idle)
                (cond .==. constE False)
         ]
  ifEq $
    counter =:=?=
      (valueVar mode .==. constE InProgress)
      .==:>.
      (inc counter)
      `elseIf`
      (valueVar mode .==. constE DurationGtEq)
      .==:>.
      (valueState counter)
      `defaultCase`
      one
  ifEq $
    res <:=?=
      ((valueVar mode .==. constE Idle)
       .||. (valueVar mode .==. constE InProgress))
      .==:>.
      (constE False)
      `defaultCase`
      (constE True)
  return res
```

Figure 2. EDSL Example

The most interesting aspect of an EDSL is the kind of new abstractions one invents for specific problem domains. As far as we know, the issue of traceability and correctness of EDSL abstractions has so far been largely overlooked. For example, the focus of the work in [4] is on ensuring correctness of the basic EDSL compiler. Our focus here is on the EDSL abstractions. How to provide sufficient information that the basic EDSL constructs can be traced to the domain-specific abstractions? How to provide evidence that the abstractions are correct?

2. EDSL Example

Figure 2 defines a temporal abstraction `durationGtEq` to check if the monitored Boolean value `condIn` (e.g. some door switch) is True for longer than the amount of cycles defined with `limitIn`. In this case the Boolean output `res` gets True. The process can start again when the input value becomes False, in which case the output becomes False too.

Function `durationGtEq` is a Haskell function and therefore we map the Haskell function parameters to some EDSL variables. EDSL primitive `node` introduces a new scope within the EDSL. This primitive is not strictly necessary but allows us to connect the variables and rules in its scope to the new abstractions, e.g. `durationGtEq`, we have built.

The `modeMachine` is a simple state machine. The three modes `Idle`, `InProgress` and `DurationGtEq` tell us if the monitored Boolean value has been true for longer than a certain amount cycles or not. Via EDSL primitive `valueState` we access the values of EDSL variables, via `constE` we turn Haskell constants into EDSL constants and via `ifEq` we represent switch-case statements. For example, the `counter` is incremented if we are in mode `InProgress`.

```
(.---->.) from to= \cond -> (from,to,cond)
modeMachine name init transitions = do
  let mmPath x = appendPath (initPath $ ModeMachine name)
                            (initPath x)
  mode <- outpVar "mode"
                  (mmPath OutVar)
  machineState <-stateVar
                  "machineState"
                  (mmPath StateVar)
                  (constE init)
  trigger <- localVar "trigger"
                    (mmPath LocalVar)
                    (undefined :: Bool)

  simpEq (mmPath InitTrigger) $
     trigger =:= constE False

  mapM_ (\(from,to,cond) -> do
     n <- freshName
     rule meta{trace = Just $ mmPath Transition} $ do
       conditions $
         (valueVar trigger .==. constE False)
         .&&. (valueState machineState .==. constE from)
         .&&. cond
       machineState =:= constE to
       trigger =:= constE True
     )
     transitions
  simpEq (mmPath SetMode) $
     mode <:= (valueState machineState)
  return mode
```

Figure 3. Modemachine and Traceability

In fact, constructs `ifEq` and `modeMachine` are themselves abstractions. That is, we use Haskell as our macro language to express these constructs in terms of more primitive EDSL constructs. As pointed out in [4], one rarely uses the primitive EDSL language. Instead, arbitrarily complex abstractions are built but ultimately the same simple code (in our case guarded actions) is generated.

However, in the context of formal software certification w.r.t safety standards such as DO-178B and IEC 61508 it is required to provide "evidence" for every single line of source generated.

3. Traceability

Our goal is to maintain precise links (traces) between abstractions specified in the macro language and the code generated by the primitive EDSL. As a running example, we consider the `modeMachine` abstraction whose encoding in terms of the primitive EDSL is given in Figure 3. We first review the principles behind the encoding of `modeMachine` and then consider the issue of traceability.

State variable `machineState` represents the current mode we are in. Each mode transition translates into a primitive `rule`. A transition fires if none of the earlier transition has fired (see `trigger`), we are in the proper `machineState` and the conditions `cond` are satisfied. Then, we execute the transition by adjusting the `machineState` and remember firing of a transition via `trigger`. Via the primitive `simpEq` (a `rule` with an always true condition) the `trigger` is reset and the current `machineState` is assigned to the output `mode`.

Traceability here means that we can precisely identify which `rule` represents a transition, a simple equation etc. In a first step, we define a data type `MMKind` in Figure 4 to represent the various elements of a `modeMachine`. In essence, `MMKind` represents the *explicit* AST of a `modeMachine`.

Primitive constructs `rule`, `simpEq`, `outpVar`, `stateVar` and `localVar` will be annotated with additional meta data to provide links to the AST elements which they shall represent. For example, consider an excerpt of Figure 3.

```
trigger <- localVar "trigger"
                    (mmPath LocalVar)
                    (undefined :: Bool)
 ...
 simpEq (mmPath InitTrigger) $
    trigger =:= constE False
```

```
class TraceCl a where
   printTrace :: a -> String
data Path = Stop
         | forall a. TraceCl a => Prefix a Path
initPath x = Prefix x Stop
appendPath Stop x = x
appendPath (Prefix x p1) p2 = Prefix x $ appendPath p1 p2

data MMKind = ModeMachine String | InitTrigger | Transition
        | SetMode | StateVar | OutVar
        | LocalVar deriving Show
instance TraceCl MMKind where
    printTrace (ModeMachine name) = "ModeMachine_" ++ name
    printTrace x = show x
```

Figure 4. Metadata

The first part declares that `trigger` is the `modeMachine`'s local variable. The second part declares that the purpose of the simple equation is to reset the trigger variable to the initial false value. Meta data information is maintained during translation to low-level C code. The thus annotated EDSL code fragment translates to

```
// Trace: Node_durationGtEq::ModeMachine_durGtEq::LocalVar
bool s_LOCAL_GENtrigger7;
...
// Rule: simpEq8
// Trace: Node_durationGtEq::ModeMachine_durGtEq::InitTrigger
if(( 1 == 1 )) {
 bool __1;
 __1 = false;
 s_LOCAL_GENtrigger7 = __1;
}
```

From the trace information in the generated code we can derive that `s_LOCAL_GENtrigger7` is the triggering variable of the `modeMachine` with name `durGtEq`. The `modeMachine` themselves is part of a bigger `node` abstraction with name `durationGtEq`.

As we can see, arbitrary abstractions can be built on top of each other. This requires us to maintain the precise trace/path through all the various abstractions we have built. Hence, when adding the meta information that `trigger` represents `LocalVar` we possibly extend an already existing trace/path. Technically, we use the `Path` data type in Figure 4 to keep track of the various EDSL abstractions. To record elements of different (AST) types along a trace/-path, constructor `Prefix` is given an existential type.

For the example in Figure 3 we generate overall 20 rules and 10 variables. Thanks to the trace information each rule and variable can be easily connected to the higher-level abstraction which gave rise to the variable/rule.

Of course, it is up the designer of the EDSL abstractions to provide the proper 'semantic links' between the abstractions and the EDSL primitives. If desired we can completely omit any meta data information. For example, in Figure 2 we only declare `stateVar "counter" (constE (1::Int))` without providing further details. If omitted we only attach basic information about the surrounding scope to the EDSL variables declared, e.g.

```
// Trace: Node_durationGtEq
int s_STATE_GENcounter3;
```

4. Testing

A test consists of a part to execute the system under test (SUT) by setting the input variables at designated cycles. Our SUT here is the program from Figure 2 where we assume that the limit parameter is a constant. Here's a simple test input specified in the Emit EDSL where we repeatedly set the Boolean input condition to true for limit+1 cycles.

```
input "alwaysTrue" $
  (repeatEmit
      (limit+1)
        (emitStep $
            in1 .=. constE True))
```

Running the above inputs against the SUT results in a trace log. We analyze this trace log via a LTL-style property EDSL. Here are two sample properties.

```
property "DurationReached" $
  always $
    (valueIn in1 .==. constE True)
       .=||=>. limit $
          (valueOut out1 .==. constE True)
```

The above property makes use of some LTL pattern/abstraction where the pattern `A .=||=>. n $ B` denotes that if `A` holds `n` steps then in the `nth` step `B` must hold. Thus, we can specify concisely that the `durationGtEq` shall yield true if for limit or more cycles the input condition is true.

The next set of properties state that the output shall be false if the input is set to true for less than limit cycles.

```
map (\i->
     property ("DurationNot"++show i) $
      always $
        (valueIn in1 .==. constE False)
          .==>. 1 $
            (valueIn in1 .==. constE True)
              .=||=>. i $
                (valueOut out1 .==. constE False))
    [1..limit-1]
```

A test suite consists of a set of inputs and LTL properties. We first run all inputs against the SUT which results in a set of trace logs. Then, we match each trace log against each LTL property. Our constructive LTL matching algorithm provides us information about which parts of the formula have been matched and if there have been any failures.

5. Practical Experiences and Lessons Learned

Industrial Projects Our approach has so far been applied in two commercial projects and one case study. In the first commercial project we built a central control application in the Defense area which is connected to about ten subsidiary devices. This is a major application consisting of about 700 input/output sensor variables and compiles down to about 2000 primitive rules out of which we generate about 30,000 lines of C code (this includes some code instrumentation for coverage and logging). The second commercial project lies in the Transportation area. The EDSLs are applied to simulate external components of a central train control system. The simulations are used to perform a hardware-in-the loop test of the train system.

In both cases, the application/simulations went through a couple of major refactoring steps due to unknown or changing customer requirements. We wouldn't have met the deadline without the flexibility of the EDSL approach and the possibility of implementing new abstractions/extensions quickly while guaranteeing their correctness.

The case study is an Automotive application and is freely available with the EDSL source code distribution.

EDSL Toolchain Evolution The EDSLs and its toolchain have evolved in various directions mainly due to customer requests.

For example, for the Defense project, the customer demanded to import the test suite to DOORS, a popular commercial requirement engineering tool. Therefore, we had to customize the meta data information and turned LTL patterns into semi-formal English sentences. The result is an automatic import into DOORS where tests are immediately linked to high-level requirements as specified in the meta data.

In the Transportation project, our first target system was .Net/C♯ which then later was changed to a real-time operating system(RTOS)/C system. For us, it was pretty straightforward to compile our primitive guarded actions to either C♯ or C. The higher-level simulation part remained unchanged.

User Experience Overall six Software/System Engineers have worked with the EDSLs and its toolchain so far. The Haskell syntax took some time to getting used to. However, once the proper EDSL abstractions are in place it was possible to write the code even for experts without Haskell knowledge.

For example, in case of the Transportation project, portions of the simulations have been coded by a domain expert which had no prior exposure to Haskell and EDSLs before.

A well-known issue is the lack of reasonable customized error messages which our approach shares with other EDSLs. In our EDSLs we make heavy use of type classes to configure for example the signal type language and overload the various EDSL constructs. This results in long and hard to read types. We are in the process of simplifying parts of our EDSLs which then yields more comprehensible types.

Comparison to Commercial Tools Industrial-strength tools such as Simulink or SCADE offer a GUI interface and offer visual editors to facilitate the communication with domain experts. Our toolchain is currently operated from the command line and it can be hard to sell to customers.

Yet we see EDSLs as a secret weapon in case of software projects which require great flexibility (e.g. rapdily changing customer requirements etc). Thanks to the nature of EDSLs, new project specific abstractions can be coded quickly and easily. Moreover, we provide an expressive test system which is fully integrated into our EDSLs.

For the success of EDSLs in this domain we can make the user experience more convenient by integrating it into a GUI and by providing connectors to established commercial tools. For example, we are currently working on automatic transformations from our EDSLs to SCADE's textual language. Thus, we can take advantage of the flexibility of the EDSL approach and at the same time we are able to integrate our approach into existing tool-chains.

Acknowledgments

We thank the reviewers for their comments.

References

[1] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[2] Tom Hawkins. Atom dsl. `http://hackage.haskell.org/package/atom/`.

[3] Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury, and Jean-Louis Colaço. Experience report: using objective caml to develop safety-critical embedded tools in a certification framework. *SIGPLAN Not.*, 44:215–220, August 2009.

[4] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Experience report: a do-it-yourself high-assurance compiler. In *Proc. of ICFP'12*, pages 335–340. ACM, 2012.

[5] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[6] RTCA/DO-178B. Software considerations in airborne systems and equipment certification. pages 31–74, 1992.

[7] Martin Sulzmann and Axel Zechner. Constructive finite trace analysis with linear temporal logic. In *Proc. of TAP'12*, volume 7305 of *LNCS*, pages 132–148. Springer, 2012.

[8] Martin Sulzmann and Axel Zechner. Model checking DSL-generated C source code. In *Proc. of SPIN'12*, volume 7385 of *LNCS*, pages 241–247. Springer, 2012.

```
  ifEq $
    counter =:=?=
      (valueVar mode .==. constE InProgress)
      .==:>.
      (inc counter)
      ‘elseIf‘
      (valueVar mode .==. constE DurationGtEq)
      .==:>.
      (valueState counter)
      ‘defaultCase‘
      one
```

translates to

```
/* ////////////////////////
// Rule: simpEq16
Trace: Node_durationGtEq::SwitchCase_15::IfInitTrigger*/
if(( 1 == 1 )) {
{
bool __1;

 __1 = false;

s_LOCAL_GENtriggerIf15 = __1;

 }

}

/* ////////////////////////
// Rule: IfEq17_18
Trace: Node_durationGtEq::SwitchCase_15::IfCase*/
if((s_LOCAL_GENmode4 == InProgress && s_LOCAL_GENtriggerIf15 == false)) {
{
bool __1;
int __2;

 __1 = true;
 __2 = ( 1 + s_STATE_GENcounter3 );

s_LOCAL_GENtriggerIf15 = __1;
s_STATE_GENcounter3 = __2;

 }

}

/* ////////////////////////
// Rule: IfEq17_19
Trace: Node_durationGtEq::SwitchCase_15::IfCase*/
if((s_LOCAL_GENmode4 == DurationGtEq && s_LOCAL_GENtriggerIf15 == false)) {
{
bool __1;
int __2;

 __1 = true;
 __2 = s_STATE_GENcounter3;

s_LOCAL_GENtriggerIf15 = __1;
s_STATE_GENcounter3 = __2;

 }

}

/* ////////////////////////
// Rule: IfEq17_Default
Trace: Node_durationGtEq::SwitchCase_15::IfDefault*/
if(s_LOCAL_GENtriggerIf15 == false) {
{
int __1;
bool __2;

 __1 = 1;
 __2 = true;

s_STATE_GENcounter3 = __1;
s_LOCAL_GENtriggerIf15 = __2;

 }

}
```

Figure 5. Translation of `ifEq`

A. Traceability Example

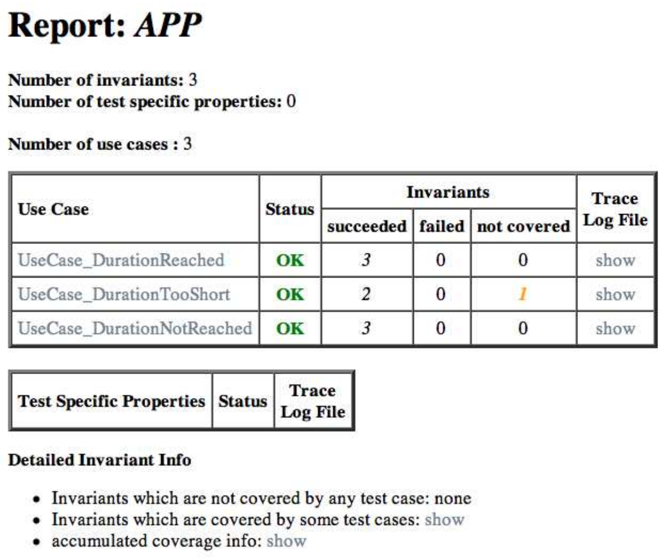
In Figure 5, we show the generated code for parts of our running example.

A.1 Test Report Example

The following description relies on color information

As a part of our EDSL toolchain we have built a 'TestRunner' tool to automate test execution. The tool generates a HTML report which contains extensive information about number of input runs, LTL properties, accumulated coverage and specific coverage when matching pairs of inputs and LTL properties. Figure 6 shows a sample test report summary. Via hyper-links the user can inspect the details.

For example, Figure 7 displays the accumulated coverage result of property "DurationReached". We find here the 'primitive' AST representation of the LTL property where all abstractions have been unrolled.



Report: *APP*

Number of invariants: 3
Number of test specific properties: 0

Number of use cases : 3

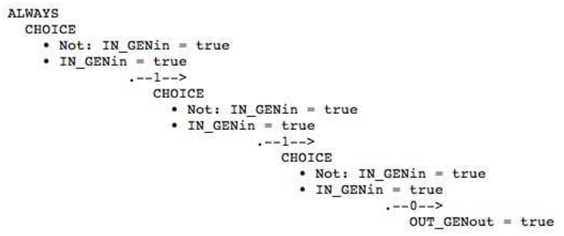
Use Case	Status	Invariants			Trace Log File
		succeeded	failed	not covered	
UseCase_DurationReached	OK	*3*	0	0	show
UseCase_DurationTooShort	OK	2	0	*1*	show
UseCase_DurationNotReached	OK	*3*	0	0	show

Test Specific Properties	Status	Trace Log File

Detailed Invariant Info

- Invariants which are not covered by any test case: none
- Invariants which are covered by some test cases: show
- accumulated coverage info: show

Figure 6. Test Report



```
ALWAYS
  CHOICE
    • Not: IN_GENin = true
    • IN_GENin = true
              .--1-->
                 CHOICE
                   • Not: IN_GENin = true
                   • IN_GENin = true
                             .--1-->
                                CHOICE
                                  • Not: IN_GENin = true
                                  • IN_GENin = true
                                            .--0-->
                                               OUT_GENout = true
```

Figure 7. Accumulated Coverage

In detail, at the Haskell level we unroll

```
property "DurationReached" $
  always $
    (valueIn in1 .==. constE True)
       .=||=>. limit $
          (valueOut out1 .==. constE True)
```

into

```
property "DurationReached" $
  always $
    (valueIn in1 .==. constE True)
       .==>. 1 $
          (valueIn in1 .==. constE True)
             .==>. 1 $
               (valueIn in1 .==. constE True)
                  .==>. 0 $
                    (valueOut out1 .==. constE True)
```

where we assume that `limit` equals 3.

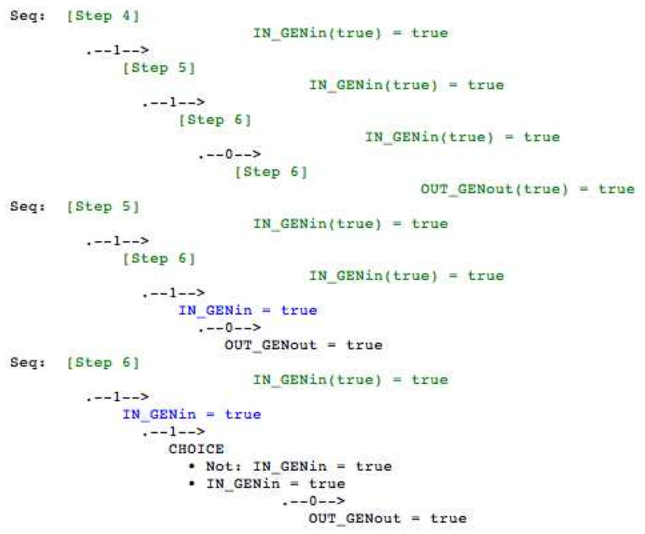
We still need one more step to reach the most elementary form. The expression

```
x .==>. 1 $ y
```

is a short-hand for

```
((.!.) x)   .\/. (x .-->. 1 $ y)
```

That is, either `x` doesn't hold or if `x` holds in the next step `y` holds.



```
Seq:  [Step 4]
                          IN_GENin(true) = true
        .--1-->
            [Step 5]
                                IN_GENin(true) = true
              .--1-->
                  [Step 6]
                                      IN_GENin(true) = true
                    .--0-->
                        [Step 6]
                                            OUT_GENout(true) = true
Seq:  [Step 5]
                          IN_GENin(true) = true
        .--1-->
            [Step 6]
                                IN_GENin(true) = true
              .--1-->
                  IN_GENin = true
                    .--0-->
                       OUT_GENout = true
Seq:  [Step 6]
                          IN_GENin(true) = true
        .--1-->
            IN_GENin = true
              .--1-->
                 CHOICE
                   • Not: IN_GENin = true
                   • IN_GENin = true
                             .--0-->
                                OUT_GENout = true
```

Figure 8. Input against LTL match

At this level, we perform the constructive LTL matching and display the matching results.

For our example, all parts are ’black’ which shows that the inputs have exercised all atomic parts of the LTL formula. This information provides some indication if the LTL properties are covered sufficiently by inputs.

Figure 8 shows the matching of the above input ”alwaysTrue” against property ”DurationReached”. In each step, the input condition is true. For brevity, we only show the matching starting with the fourth step. The result of matching an input (i.e. the trace implied) against a LTL formula is represented by displaying the parse tree. The color ’green’ highlights the matched parts whereas color ’blue’ indicates that at this point the trace ended prematurely and a full parse tree couldn’t be build. The color ’red’ highlights a violation which is not the case here.

We are in the process of improving the display of test results by mapping results at the most basic level to the EDSL abstraction level.