

Early Fault Detection in DSLs Using SMT Solving and Automated Debugging^{*}

Sarmen Keshishzadeh¹, Arjan J. Mooij², and Mohammad Reza Mousavi³

¹ Eindhoven University of Technology, Eindhoven, The Netherlands

² Embedded Systems Innovation by TNO, Eindhoven, The Netherlands

³ Center for Research on Embedded Systems, Halmstad University, Sweden
s.keshishzadeh@tue.nl, arjan.mooij@tno.nl, m.r.mousavi@hh.se

Abstract. In the context of Domain Specific Languages (DSLs), we study ways to detect faults early in the software development cycle. We propose techniques that validate a wide range of properties, classified into basic and advanced. Basic validation includes syntax checking, reference checking and type checking. Advanced validation concerns domain specific properties related to the semantics of the DSL. For verification, we mechanically translate the DSL instance and the advanced properties into Satisfiability Modulo Theory (SMT) problems, and solve these problems using an SMT solver. For user feedback, we extend the verification with automated debugging, which pinpoints the causes of the violated properties and traces them back to the syntactic constructs of the DSL. We illustrate this integration of techniques using an industrial case on collision prevention for medical imaging equipment.

Keywords: Early Fault Detection, Formal Verification, Domain Specific Language (DSL), Satisfiability Modulo Theories (SMT), Delta Debugging.

1 Introduction

Domain specific languages (DSLs, [20,15]) are used to specify software at a higher level of abstraction than implementation code, and to mechanically generate code afterwards. By trading generality for expressiveness in a limited domain, DSLs offer substantial gains in ease of use compared with general-purpose programming and specification languages in their domain of application [15]. Hence, DSLs bring formality closer to domain requirements.

Our goal is to investigate ways to provide early fault detection (see, e.g., [11]) when developing industrial software using DSLs. Program verification techniques often focus on implementation code, and heavily depend on abstraction techniques. Since DSLs are based on domain specific abstractions, we aim to integrate verification at the level of the DSL, i.e., before generating any code.

^{*} This research was supported by the Dutch national program COMMIT and carried out as part of the Allegio project.

Meta-modelling frameworks, such as the Eclipse Modelling Framework (EMF, [19]), XText [7], and MontiCore [14], provide support for developing editors, performing validation, and generating code. The validation for DSLs often concerns basic validation, such as syntax checking, reference checking, and type checking. In this paper, we focus on techniques for more advanced kinds of validation.

Our investigation is based on a prototype DSL for collision prevention, developed in collaboration with Philips Healthcare; see [16]. The main objective of this DSL is to facilitate the reuse of software among different product configurations. The primary goals are hence to reach a convenient abstraction level, and to generate implementation code. Since correct and timely functioning is vital for medical systems, this prototype DSL is an interesting study case for advanced validation.

Through our interaction with the software developers, we have identified two important user requirements for the integration of advanced validation in industrial DSLs. These have guided our selection of formal techniques.

The first requirement is to hide the validation techniques from the user of the DSL. This implies that a push-button technology should be used, such as model-checking [1] or satisfiability checking [3]. It also implies that we should not rely on user knowledge about applying verification techniques and analyzing their outputs. To this end we mechanically generate the validation input from the DSL instance; this input includes both the formal model and the formal properties. We also translate any property violations back to the abstraction level of the DSL. To detect the syntactic constructs that cause the property violations, we have used an automated debugging technique called delta debugging [23,4,22]. Thus the detected causes are presented in the DSL editor.

The second requirement is to provide feedback to the users in a short amount of time (in the order of seconds to minutes). This often rules out model-checking techniques based on explicit state-space exploration [9], and generic numerical analysis techniques for hybrid systems [8]. We aim to use existing tools as they are, and therefore we refrain from developing ad-hoc abstraction techniques for our specific DSL. We have used Satisfiability Modulo Theories (SMT) [2,6] solving. SMT solvers check satisfiability of first order logic formulae with respect to a combination of background theories, e.g., on integer arithmetic. In recent years, SMT solvers have been extensively applied as an efficient means for program verification [6].

Thus we propose an integration of three techniques, viz., domain-specific languages, SMT solving and delta debugging. Fig. 1 gives an overview of our approach; we refer to it throughout this paper. The traditional use of DSLs is depicted at the left, starting with a system specification which is formalized as a DSL instance. The DSL instance is used for basic validation, and for generating implementation code in languages such as C++. In addition, we introduce advanced validation by automatically generating a set of SMT problems that express some system properties for the DSL instance. Finally the verification results are linked back to the DSL instance.

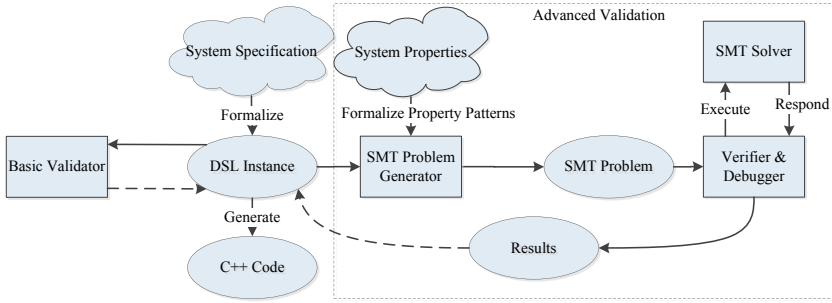


Fig. 1. Overview of the Automated Approach

Although the SMT problems are generated from the DSL instance, the verification results do not change the DSL instance; so Fig. 1 is not a round-trip engineering environment. Instead, the verification results are displayed in the DSL editor. Automated debugging is used to determine a fault location.

Related work. Delta-debugging has initially been developed for debugging programs. We are aware of a few research works [12,21] that apply this technique to more abstract domains. In this paper, we apply it to a declarative DSL.

Integrations of satisfiability checking and debugging have been studied in both hardware and software domains. [18] applies such an integration in the context of logic circuits. [13] proposes a method that, given a C program with an assert statement and a set of failing test cases, provides a list of potential fault locations in an interactive process. This method analyzes a failure trace by encoding the definition and use relation for program variables as MAX-SAT problems. Unlike C programs there is no definition-use relation among the statements of our DSL. Hence, this approach is not applicable in our case.

An integration of verification techniques and DSLs is reported in [17]. Their goal is to maximize reusability among different DSLs. They extract commonalities shared between different DSLs (e.g., a Boolean expressions module) and encapsulate them as analysis-DSLs. Analysis tools, such as model checkers and SMT solvers, are applied to instances of analysis-DSLs. Their validation is limited to properties shared between various domains, e.g., completeness of a set of specified restrictions, and consistency of simultaneously activated restrictions.

Overview. In Section 2, we introduce the industrial prototype DSL, its syntax and informal semantics. Subsequently, we describe the kinds of properties that we aim to validate in Section 3. The translation to SMT is presented in Section 4. Automated debugging for determining the causes of property violations is presented in Section 5, whereas the integration with the DSL editor is reported in Section 6. In Section 7 we draw some conclusions and suggest future research.



Fig. 2. Industrial Study Case

2 A Prototype DSL for Collision Prevention

To illustrate our approach, we consider the interventional X-ray scanners of Philips Healthcare. These systems consist of several moving objects with shapes as sketched in Fig. 2(a). For example, the Table can be moved horizontally, the Detector can be moved vertically, and the CArm can be rotated.

To prevent collisions between these objects, the architecture contains a safety layer as depicted in Fig. 2(b). All movement requests from the user to the motors pass this layer. For making decisions on user requests, this layer stores data from the sensors in internal structures called “geometric models”. In particular each geometric model stores the (shortest) distance between each pair of objects.

To describe the safety layer, we consider a simplified prototype DSL that focuses on decision rules for collision prevention. We illustrate the syntax and the intuitive meaning of the syntactic constructs using the example instance in Fig. 3. For confidentiality reasons, numbers and details have been changed.

2.1 Physical Objects and Geometric Models

Each DSL instance declares the physical objects in the system. The example in Fig. 3 corresponds to the geometry in Fig. 2(a) with three objects, viz., Table, CArm and Detector. The shapes of the objects are not specified in the DSL.

This example DSL instance declares a *predefined* geometric model and a *user-dependent* geometric model:

- Actuals: current object distances, as given by the sensors;
- LookAhead: predicted object distances, based on Actuals and user requests.

The definitions of these models are internal, and not specified in the DSL.

2.2 Movement Restrictions

The user requests consist of a vector for each object movement (translation and rotation). The collision prevention logic is specified in terms of restrictions on these object movements. Each restriction contains an activation condition, which

```

// --- Context Declarations -----
object Table
object CArm
object Detector

model Actuals           predefined
model LookAhead        userdependent

// --- Restrictions -----
restriction ApproachingTableAndCArm
activation
  Distance[Actuals](Table, CArm) < 35 mm + 15 cm
effect
  absolute limit CArm[Rotation]
  at ((Distance[Actuals](Table, CArm) - 35 mm) / 15 cm) * 10 dgps

restriction ApproachingTableAndDetector
activation
  Distance[LookAhead](Table, Detector) < 35 mm + 15 cm
  && Distance[LookAhead](Table, Detector) <
  Distance[Actuals](Table, Detector)
effect
  relative limit Detector[Translation]
  at ((Distance[LookAhead](Table, Detector) - 35 mm) / 15 cm)

```

Fig. 3. Example Instance of the DSL

is a boolean expression, and an effect that is only considered when the activation condition evaluates to true. The effect specifies a speed limitation for a specific object movement; to be more precise, a limitation on the (Euclidean) norm of the movement vector. An *absolute* speed limit specifies a maximum speed that may be requested to the motors. A *relative* limit indicates the maximum percentage of the user request that may be requested to the motors.

The example restrictions in Fig. 3 illustrate that the expressions can refer to the (shortest) distance between two objects in a specific geometric model. Constants can be annotated with measurement units, or otherwise a default unit is assumed. Further processing of a DSL instance unifies the applied units.

For each object movement, multiple restrictions can specify absolute and relative limits. In this case, for each object movement, only the most-restrictive activated limits are considered, i.e., the minimum of the absolute limits and the minimum of the relative limits; the other limits are masked. Given the incoming request vector $\overrightarrow{inRequest}$ for an object movement, we first compute the requested speed $inSpeed$. Using the most-restrictive activated limits $absLimit$ and $relLimit$ for this object movement, we compute the resulting speed $outSpeed$ and the outgoing movement request vector to the motors $outRequest$ as follows:

$$\begin{aligned}
 inSpeed &= \mathbf{norm}(\overrightarrow{inRequest}) \\
 outSpeed &= \mathbf{min}(absLimit, relLimit \times inSpeed) \\
 \overrightarrow{outRequest} &= \frac{outSpeed}{inSpeed} \times \overrightarrow{inRequest}
 \end{aligned}$$

3 Validation Properties

In this section we describe several kinds of properties, that can be analysed early in the software development cycle, in particular before generating code.

3.1 Basic Validation

Practically all modern editors for programming languages and domain-specific languages offer some basic types of validation:

- based on the language (context-free analysis):
 - parsing: syntactic constructs are in accordance with the DSL grammar;
- based on the parse tree (context-dependent analysis):
 - referencing: references refer to elements that have been defined;
 - type checking: expressions have a well-defined type.

In addition, there can be domain-specific constraints like acyclic dependencies. There can also be warnings for correct fragments that are probably not intended, such as, in our DSL, the distance between an object and itself.

3.2 Advanced Validation

Our aim is to offer validation that goes beyond basic validation. In this section we consider the system properties focusing on collision prevention, which include value ranges, safety properties, and absence of deadlocks.

In our example DSL, such checks often require additional knowledge about the environment, including the geometric models and the timing. We try to keep these details to a minimum in order to make the verification feasible and to give quick feedback to the user. In our analyses this has an impact on the following:

- distances: We only assume that the distance function on pairs of objects is symmetric and gives non-negative values. We ignore whether the distances are feasible in practice.
- timing: We ignore the acceleration characteristics of the physical objects, and any time delays between sensing and acting.

However, this can result in false positive reponses for well-definedness of expressions and safety properties and false positive/negative responses for deadlock. The challenge is to balance the number of false positive/negative results with the number of additional details that need to be provided. In what follows, we categorize the kind of checks that could be useful for our DSL users.

Well-definedness of Expressions. There are some general conditions that can be checked. For example, a potential division by zero, or a potential exponentiation resulting in a complex number. (Similarly, for DSLs allowing for case analysis, we can check whether the cases are complete and non-overlapping.) Such checks are more involved than basic type checking, because they involve the valuation of distance variables and arithmetic operations on them.

Ranges. The minimum of activated absolute limits of each object movement should be a non-negative real number; similarly, the minimum of activated relative limits should be a real number between 0 and 1.

Safety. The ultimate goal of the safety layer is to prevent collisions. We check specific speed limits when two objects are “very close” and “approaching”. We also check monotonicity properties (with respect to each distance parameter), e.g., the closer two objects, the stricter the speed limits. The notion of “approaching” can be expressed by comparing object distances in the Actuals (current distances) and the LookAhead (predicted distances) geometric models; see also the activation condition of `ApproachingTableAndDetector` in Fig. 3.

Deadlock. Sometimes objects can reach a deadlock position. Consider for example restriction `ApproachingTableAndCArm` in Fig. 3. Suppose we move the Table and the CArm towards each other. If the remaining distance is exactly 35 mm, then the speed of the CArm is limited to 0, independently of any (future) user request for the CArm. Unless there is another way to move the CArm, this object has reached an individual deadlock.

We aim to warn the DSL-user for such situations, where certain sensor inputs can stop an object independently of any future user request. As we abstract from the dependencies between distance parameters in different geometric models, our possibilities to formulate this property are limited. We formulate it as “for each object, and for each valuation of the Actuals geometric model, there exist a valuation of the LookAhead geometric model (a user dependent geometric model), such that the object can move”. This can result in false positive/negative responses. The false negative responses may sound serious in our context, but this check is still useful as a warning for typical domain errors.

4 From DSL Instances and Properties to SMT

In this section we describe the SMT problem generator from Fig. 1. We describe the advanced validation properties from Section 3.2 using examples, but for each property also a formal pattern is defined. Given any DSL instance, these properties are mechanically instantiated to a set of SMT problems in the common SMT-LIB format, which is supported by various SMT solvers.

In Section 4.1 we address well-definedness of expressions, and in Section 4.2 we address the other properties, which need to take all restrictions into account. Finally, in Section 4.3 we report on our experiences with SMT solvers.

Note that all SMT expressions are written in the prefix style. As a convention, in our examples any SMT variable `GeoModel_Object1_Object2` represents the expression `Distance[GeoModel](Object1,Object2)` in the DSL instance. So we can assume that `GeoModel_Object1_Object2` is non-negative. The SMT problem generator (Fig. 1) guarantees that `Distance[GeoModel](Object1,Object2)` and `Distance[GeoModel](Object2,Object1)` are represented by the same variable. For brevity we use `ahead` instead of `LookAhead` in our naming convention.

4.1 Well-definedness of Expressions

Since users can specify complicated activation conditions or speed limits, we provide mechanisms to warn for mathematically undefined expressions. As an example we focus on potential divisions by zero, which can occur at two places. First, any divisions in the activation condition are checked in isolation. Second, divisions appearing in effect clauses are checked under the assumption that the corresponding activation condition holds.

Consider the following restriction which contains division at both locations:

```
restriction DivByZeroSample
  activation 1 / (1 + Distance[Actuals](Table,CArm)) > 0 &&
             Distance[Actuals](Table,CArm) < 5
  effect absolute limit CArm[Rotation] at
             1 / (6 - Distance[Actuals](Table,CArm))
```

Assuming non-negative distances, both checks are satisfied in our example. The following assertion statement encodes the check for the effect clause in SMT.

```
(assert (forall ((actuals_Table_CArm Real))
  (implies (and (>= actuals_Table_CArm 0.0)
                (> (/ 1.0 (+ 1.0 actuals_Table_CArm)) 0.0)
                (< actuals_Table_CArm 5.0) )
    (not (= (- 6.0 actuals_Table_CArm) 0.0)) )))
```

In this example, the SMT variable `actuals_Table_CArm` corresponds to `Distance[Actuals](Table,CArm)`, and condition `(>= actuals_Table_CArm 0.0)` encodes the domain knowledge that the used distances are non-negative.

4.2 Ranges, Safety, and Deadlock

The remaining properties need to take all restrictions into account. We first introduce a procedure to translate the speed limits enforced by the restrictions to SMT expressions. Each restriction is mapped to a single SMT expression, and afterwards they are combined. This allows for tracing the detected faults back to the corresponding DSL constructs in Section 5. For each identified pattern for these properties we give an example from Fig. 3. To keep the formulae simple, we omit the information that each distance is non-negative (see Section 4.1).

Consider the following general template of a restriction:

```
restriction [restriction]
  activation [act_restriction]
  effect
    relative/absolute limit [object_movement] at [eff_restriction]
```

Each `restriction` is translated to a function definition with as parameters the distances it depends on. We encode restrictions as functions with an `if-then-else` (`ite`) structure with `[act_restriction]` and `[eff_restriction]` specified as the condition and the `then` part of the conditional statement, respectively.


```
(define-fun func_restriction ((arg_1 Real)...(arg_n Real)) Real
  (ite [act_restriction]
      [eff_restriction]
      infinity ))
```

We define a sufficiently large number as **infinity**. If the activation condition is not satisfied, then **infinity** is returned, implying that there is no speed limit.

Multiple active restrictions can affect the same absolute/relative limit of an object movement. In this case, if at least one of the effects is active, we take the minimum of the activated effects as the overall effect for this limit. Otherwise, there is no restriction on the object movement. In SMT, the overall effect is specified again as a function with an **ite** structure. The parameter set of this function is the union of the parameter sets of the contributing functions.

```
(define-fun Object_Movement_Limit ((arg_1 Real)...(arg_n Real)) Real
  (ite (or [act_restriction_1] [act_restriction_2] ...)
      (min (func_restriction_1 arg_11 arg_12 ... arg_1k)
          (func_restriction_2 arg_21 arg_22 ... arg_2l)
          ... )
      ([DEFAULT_VALUE]) ))
```

The value of **[DEFAULT_VALUE]** is determined by the limit type. For relative limits, 1 is used; for absolute limits, **infinity** is used.

Let **RelDetTrans** be the SMT function that specifies the overall relative translation limit for *Detector*. In our example, the only restriction that contributes to this overall limit is **ApproachingTableAndDetector**, which is specified in terms of the two parameters **actuals_Table_Detector** and **ahead_Table_Detector**. These are also the parameters of the overall function **RelDetTrans**.

Ranges. For functions specifying the overall relative limit we check that the return value is between 0 and 1 for any valuation of distance parameters. The pattern for absolute limits is similar. For example, the following property will be generated for Fig. 3. Given the function **RelDetTrans** that specifies the overall relative translation limit for *Detector*, this property specifies that the relative limit for the *Translation* movement of *Detector* is at most 1:

```
(assert (forall((actuals_Table_Detector Real)(ahead_Table_Detector Real))
  (<= (RelDetTrans actuals_Table_Detector ahead_Table_Detector) 1.0) ))
```

Safety. As an example, we consider the monotonicity properties for each relative/absolute limit and each rotation/translation movement with respect to each distance parameter. Based on Fig. 3, the following property is generated to verify the monotonicity of the relative limit of the translation movement for *Detector* with respect to **actuals_Table_Detector**. This means that decreasing this distance parameter, while maintaining the other distance parameters, may not lead to a more relaxed limit.

```
(assert (forall ((actuals_Table_Detector Real)(ahead_Table_Detector Real)
  (actuals_Table_Detector' Real))
```

(implies

```
(<= actuals_Table_Detector actuals_Table_Detector')
(<= (RelDetTrans actuals_Table_Detector ahead_Table_Detector)
(RelDetTrans actuals_Table_Detector' ahead_Table_Detector)) )))
```

Deadlock. We identified the following pattern to check for absence of rotation/translation deadlock: “for each valuation of distance parameters in Actuals, there exists a valuation of LookAhead (a user-dependent geometric model) such that relative and absolute limits are non-zero”.

For the example in Fig. 3, the following property expresses deadlock freedom of the *Detector* translation movement. There is no absolute limit specified for this movement and this property only depends on the arguments of *RelDetTrans*.

```
(assert (forall ((actuals_Table_Detector Real))
(exists ((ahead_Table_Detector' Real))
(not (= (RelDetTrans actuals_Table_Detector ahead_Table_Detector') 0.0))
)))
```

4.3 Feasibility of SMT Solving

Applying this translation to real examples has led to some observations. First of all, most state-of-the-art SMT solvers have limited support for non-linear constraints such as exponentiation. Thus the occurrence of complex non-linear expressions in a DSL specification may limit the analysis power of our method. In our examples, exponentiation was mainly applied to model brake patterns. We have temporarily isolated these patterns from the rest of the DSL. Approximating non-linear constraints remains as one of the issues that we want to investigate in our future work.

Secondly, in order to keep validation practically feasible, we have slightly modified the SMT expressions. Since **forall** is an expensive operation for SMT solvers [10], we follow a counterexample-based approach. Instead of showing that the expressions hold for all parameter values, we aim to find parameter values that violate the property; in other words, if the negated property cannot be satisfied by any valuation, the property itself holds for all possible valuations. We have not used specific facilities (such as quantifier instantiation) provided by specific SMT solvers (such as Z3) in our analyses.

5 Automated Debugging

Based on the SMT problems presented in Section 4, the “verifier and debugger” component in Fig. 1 checks the validity of the properties. Since we encode the DSL restrictions as SMT functions with distance parameters, for any violated property, SMT solvers provide a counterexample in terms of distance values.

We aim for a debugger that mechanically computes the location of any fault in terms of the DSL instance. In this section we first describe suitable locations to report faults for the different types of properties. Then we present a procedure to compute these locations. Finally we discuss how to avoid computing masked restrictions as locations.

5.1 Fault Location

In case of any property violation, we aim to indicate the location of any fault in the DSL instance. We distinguish three kinds of locations in the DSL:

Expression. The well-definedness property from Section 4.1 is defined for each expression in isolation. In case of a violation, the fault location is the expression itself.

Restriction. The properties from Section 4.2 are verified against the whole DSL instance. In these cases the fault locations are the restrictions that can be pivotal in causing the violation. We define this as follows:

“A restriction r is a pivotal restriction for causing the violation of property P , if there exists a set of restrictions that does not violate property P , but after adding restriction r the property is violated.”

Fixed. If a property is violated for all subsets of the restrictions, then there is no pivotal restriction. For our example properties, this can apply to safety properties that specify a certain speed limit. For such properties that do not (trivially) hold for the empty set of restrictions, we report any violations at a fixed location in the DSL instance.

Debugging is only needed when restrictions should be identified as fault location. In the remainder of this section, we focus on the properties from Section 4.2 that are trivially valid for the empty set of restrictions and violated by the full set of restrictions.

5.2 Procedure to Locate a Single Pivotal Restriction

Our debugging procedure is based on the delta-debugging approach of [22]. In [22] the delta-debugging procedure is introduced for isolating the relevant part of a failure inducing program input. We adapt this procedure to our setting to detect restrictions that cause property violations. In particular, for a violated property we aim to find a pivotal restriction by narrowing down the difference between sets of passing (satisfying the property) and failing (violating the property) restrictions. Our procedure can be summarized as follows:

1. Choose a passing (R^+) and a failing (R^-) set, i.e., a set of restrictions that satisfies the property and a set that violates the property, such that $R^+ \subseteq R^-$. We choose R^+ as the empty set, and R^- as the set of all restrictions.
2. Repeatedly try to minimize the difference between sets R^+ and R^- :
 - (a) Select a set R of restrictions such that $R^+ \subset R \subset R^-$;
 - (b) Use the SMT solver to check whether R satisfies the property;
 - (c) If set R satisfies the property, then replace the passing set R^+ by R , otherwise replace the failing set R^- by R .
3. The single restriction r that distinguishes the passing set R^+ from the failing set R^- is a pivotal restriction for the property violation.

			Step 2.(a)	Step 2.(b)	Step 2.(c)	Step 3
Iteration	R^+	R^-	R	Status of R	Minimization	Fault
1	{}	$\{r_1, r_2\}$	$\{r_1\}$	satisfies the property	$R^+ := R$	-
2	$\{r_1\}$	$\{r_1, r_2\}$	-	-	-	r_2

Fig. 4. Isolating a faulty restriction with delta-debugging

As an example, consider Fig. 3 where the relative limit for Detector translation can be negative. Fig. 4 illustrates the application of this fault isolation procedure to detect a faulty restriction. Restrictions r_1 and r_2 represent the first and second restriction in Fig. 3. Finally, in the second iteration, restriction r_2 , i.e., **ApproachingTableAndDetector**, is identified as a fault location.

From the description of Step 2(a) one can easily deduce that the fault isolation procedure is non-deterministic. In the presence of multiple faulty restrictions, each execution of this procedure can identify a different restriction.

Regarding the performance, in the worst case the number of iterations of Step 2 is linear in the total number of restrictions. One can constrain the choice of R in Step 2(a) to make it logarithmic. Moreover, the debugging considers only subsets of the original specification, for which SMT solving has a lower complexity (i.e., typically consuming much less time and memory).

5.3 Masked Restrictions

The procedure from Section 5.2 can also report restrictions as faults at points where they are masked (see the DSL semantics in Section 2.2). To illustrate this, we consider three example restrictions r_1 , r_2 , and r_3 based on a single distance parameter. Fig. 5(a) represents the individual relative limits for a specific object movement in terms of the distance parameter. The overall effect is defined as the minimum of the individual effects, as depicted in Fig. 5(b).

The effect of a restriction r is masked for a given set of distance values, if there exists at least one restriction r' for which the effect is less than the effect of r for the same combination of distance values. In this example, the effect of restriction r_3 is masked by another restriction for every distance value.

Considering the range property “relative limits should be at most 1”, restriction r_3 in isolation violates this property, and hence the procedure from Section 5.2 can indicate this masked restriction as the fault location. Masking is no issue for the verification, but it is undesired that debugging reports masked restrictions as fault location.

If the semantics of the DSL is correctly implemented throughout code generation, masked restrictions will never lead to failures and hence, are considered spurious by the domain experts. To avoid reporting masked restrictions as fault locations, we replace all restrictions by just their unmasked parts, as shown in Fig. 5(b). This requires a small modification of the SMT formulations from Section 4.2. For restriction r_3 it results in the following SMT expression:

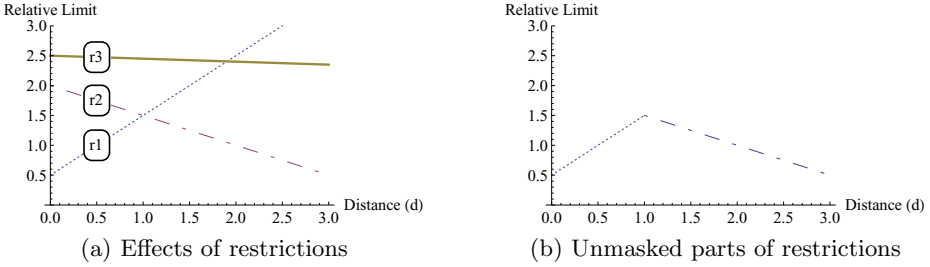


Fig. 5. Masked restriction

```

(define-fun r3 ((d Real)) Real
  (ite (and [act_r3]
            (not (and [act_r1] (< [eff_r1] [eff_r3]))))
        (not (and [act_r2] (< [eff_r2] [eff_r3]))))
        [eff_r3]
        infinity ))

```

In comparison with the encoding from Section 4.2, the activation condition is extended with two conjuncts indicating that its effect is not masked by an active restriction $r1$ nor by an active restriction $r2$. We apply a similar encoding to restrictions $r1$ and $r2$. In this way masked restrictions have no effect any more, and hence, they cannot be identified as fault location.

6 Integration with DSL Editor

We have implemented the introduced verification and debugging approach using the Eclipse Modeling Framework (EMF, [19]). Xtext is the open-source framework that we have applied to specify the grammar of the DSL. It is integrated with Xtend for validation and code generation. Z3 [5] is the SMT solver that we have used in our experiments. To hide all the verification and debugging strategies from the user, we provide the user with a Python script that for a given DSL instance verifies the set of predefined properties through a sequence of calls to Z3. For any violated property the debugging procedure is automatically invoked.

Basic validators are continuously executed while editing an instance of the DSL. To avoid additional delays while editing, we have decided not to perform continuous validation using SMT checkers. We generate the SMT problems and the Python script using an Xtend code generator. The validation can be initiated on user request by invoking the Python script. The validation results are stored, interpreted by a validator and shown back in the editor.

The user is notified about the validation results using “warnings”, which result in a yellow underlining of the problematic parts together with a textual message; see Fig. 6. We cannot use “errors”, because they block future executions of all code generators (including the SMT problem generator). We also warn the user about verification or debugging attempts for which the corresponding SMT problem is not decidable (e.g., as a result of non-linear expressions).

```

restriction ApproachingTableAndCArm
activation
  Distance[Actuals](Table, CArm) < 35 mm + 15 cm
effect
  absolute limit CArm[Rotation]
  at ((Distance[Actuals](Table, CArm) - 35 mm) / 15 cm) * 10 dgps

```

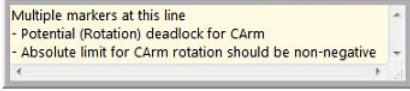


Fig. 6. Debugging results displayed in the DSL editor

7 Conclusions and Future Work

We have used a Domain Specific Language (DSL) for collision prevention to study ways to support early fault detection in industrial applications. The goal is to add value to the use of DSLs beyond code generation. In particular we have focused on validation types that are more advanced than the usual basic types of validation that can be found in modern programming environments.

For this prototype DSL, we have shown a useful set of advanced properties that can be verified efficiently using the SMT solver Z3. Actual instances consisting of 16 distance parameters from geometric models, and 81 restrictions lead to 264 generated properties from 5 property patterns. In case of 226 violations, the whole advanced validation process (including a non-optimized generator of SMT problems and Python scripts (22 sec.), and verification and debugging (105 sec.)) takes about 2 minutes on a standard desktop computer. The results are displayed at logical locations in the DSL editor. To this end, we have integrated three techniques, viz., domain-specific languages, SMT solving and delta debugging.

In the studied DSL, restrictions can sometimes be masked by other restrictions and hence they have no observable effect. In particular, we have shown how to ensure that masked restrictions are not reported as fault location.

We envisage some possible extensions of the present work. The debugging procedure can be extended to detect all possible causes of a property violation. Moreover, we aim to investigating other abstraction levels in order to rule out false positive/negative responses.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, vol. 185, pp. 825–885 (2009)
3. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
4. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of ICSE 2005, pp. 342–351. ACM (2005)

5. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. *Communications of the ACM* 54(9), 69–77 (2011)
7. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: SPLASH/OOPSLA Companion, pp. 307–309. ACM (2010)
8. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
9. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
10. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
11. Hooman, J., Mooij, A.J., van Wezep, H.: Early fault detection in industry using models at various abstraction levels. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 268–282. Springer, Heidelberg (2012)
12. Hwang, J.H., Xie, T., Chen, F., Liu, A.X.: Fault localization for firewall policies. In: Proceedings of SRDS 2009, pp. 100–106. IEEE Computer Society (2009)
13. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. *ACM SIGPLAN Notices* 46(6), 437–446 (2011)
14. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. *J. STTT* 12(5), 353–372 (2010)
15. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Computing Surveys* 37(4), 316–344 (2005)
16. Mooij, A.J., Hooman, J., Albers, R.: Gaining industrial confidence for the introduction of domain-specific languages. In: Proceedings of IEESD, 2013 (to appear, 2013)
17. Ratiu, D., Voelter, M., Molotnikov, Z., Schaetz, B.: Implementing modular domain specific languages and analyses. In: Workshop on MoDeVVa (2012)
18. Smith, A., Veneris, A., Ali, M.F., Viglas, A.: Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24(10), 1606–1621 (2005)
19. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. Pearson Education (2008)
20. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. *SIGPLAN Notices* 35(6), 26–36 (2000)
21. Woehrle, M., Bakhshi, R., Mousavi, M.R.: Mechanized extraction of topology anti-patterns in wireless networks. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 158–173. Springer, Heidelberg (2012)
22. Zeller, A.: Why Programs Fail? A Guide to Systematic Debugging. Morgan Kaufmann (2009)
23. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28(2), 183–200 (2002)