SARL: Framework Modeling for Static Analysis

Luca Negrini
Università di Verona and JuliaSoft SRL
Verona, Italy

Pietro Ferrara
JuliaSoft SRL
Verona, Italy

Abstract

Modern object-oriented applications make heavy use of *frameworks* like AspectJ and ASP.NET. These frameworks not only provide some library code, but they often extend and modify the execution model of the object oriented software. Therefore, a precise static analysis on such applications needs to take into account such execution models, since each framework may modify either the structure and/or the behavior of the application under analysis. In this paper we introduce SARL, a domain-specific language that allows to specify the behaviors of frameworks. Such language can be applied to object-oriented programs using these frameworks to annotate some components in order to model the framework's behavior. The experimental results show that the number of false alarms produced by an industrial static analyzer (Julia) can be greatly decreased using SARL.

1 Introduction

Static analysis allows to prove properties of computer programs without executing them. Such properties vary from the absence of runtime errors, to functional correctness and detection of illicit information flows. This is a very relevant topic nowadays, since most industrial software deals with users' sensitive data that must not be disclosed, or run in potentially hostile environments that could attack it if the software is vulnerable (e.g., through SQL injections).

However, most software relies on external frameworks (that is, third party libraries that provide an ad-hoc model of execution) like AspectJ [1] and ASP.NET [2], avoiding the reimplementation of common code in favor of reusable and highly tested code written by more expert programmers. The focus of such frameworks may be either (i) to provide correct implementations of various functionalities, (ii) to automatically generate useful code, or (iii) to give the program a specialized structure that will be exploited at runtime in order to obtain particular functionalities. In each of the above scenarios, a static analyzer may end up raising alarms on auto-generated code or on inherited design patterns. For instance, a static analyzer without any specific knowledge on ASP.NET would raise deadcode alarms on all the event handlers, since these methods are not accessible to external components and not explicitly called by the given application. Such alarms are therefore considered as an imprecision by the analyzer's end users, while they are real unsoundness of the analysis w.r.t. the real execution model of the application. There are by now dozens of different frameworks (with new ones appearing every month), and usually static analyzers are not aware of their semantics, thus producing an overwhelming amount of false alarms and/or missing to analyze some parts of the code.

In this paper we present SARL (Static Analysis Refining Language), a domain-specific language that is aimed at instructing static analyzers about the execution model of a framework, thus *refining* their analyses and improving their industrial usefulness. SARL is applicable to software written in statically type-safe, object-oriented programming languages (e.g., Java and C#).

SARL has been implemented in Julia [11], an abstract interpretation based semantic static analyzer. Julia analyses bytecode obtained from the compilation of programs written in Java, Android and C#. In Julia, annotations play a crucial semantic role: they instruct the analysis about the structure and the expected behavior of an application, and provide information between analysis components. From a framework perspective, they can be used to model its behaviors. The most relevant Julia's annotations are (i) `EntryPoint` specifies that a method might be an entry point (that is, called externally by the framework) of the application execution, and (ii) `ExternallyRead` and `Injected` specify that a field could be read or written (with arbitrary values) by the framework, respectively. Such annotations could be also adopted by the user to manually specify the behavior of the program.

The main idea of SARL is to allow one to concisely produce a set of rules, called *framework specification*, that describes how applications relying on a framework might be executed. Each rule specifies the conditions required to annotate a program member (e.g., filed or method) with a specific kind of annotation. Such a specification can be then automatically applied to any application in order to produce a collection of annotations whose semantics is already understood by Julia.

2 The SARL language

A SARL's framework specification is built of five components: (i) rules, (ii) implications, (iii) specifications, (iv) predicates, and (v) library specifications. The main focus of a specification is to add annotations to the code under analysis to represent the execution model of a framework on a specific application. Hence, each component (except for rules) aims at defining the necessary conditions to apply specific annotations to program's members.

2.1 Conditions

Conditions are the basic blocks to define when a specification should be applied. In particular, a condition may be (i) the application of a predicate (via its name), (ii) a logical operator (binary *and*, binary *or*, unary *not*) applied to other conditions, (iii) a non-terminal condition (that is, a condition that does not lead to a Boolean value, but has to be followed by another condition), or (iv) a terminal condition (that is, a condition that compares a value to a constant). A condition may target an annotation, a class, a field, a method, a method's parameter, or a method's local variable. Each of these targets may be queried for one of its properties, like name, type or annotations.

For instance, line 18 of the ASP.NET specification in Figure 2 specifies to apply the annotation if the component satisfies the predicate *isNestedComponent*.

2.2 Rules

A rule defines a condition to be satisfied in order to apply the specification. It can be either (i) an *analysis* rule, defining the object-oriented language to whom the framework might apply (e.g., .NET or Java), or (ii) a *code* rule, defining what should be found inside the target application. The use of a framework can usually be identified by the use of specific types as supertypes, or the use of some annotations from the library. Hence, two types of code rules have been defined: *superclass* rules and *annotation* rules, each checking if the corresponding concept is used inside the application. In addition, we introduced the *always* rule, that serves as an always-true code rule, to allow one to apply a specification to any analysis targeting a programming language.

Consider for instance the specification of AspectJ reported by Figure 1. The first two lines report two rules specifying that the specification should be apply if and only if (i) we are dealing with a Java application (line 1), and (ii) there is at least one annotation whose signature starts with *org.aspectj* (line 2).

2.3 Implications

An implication lets one bind an *implying* annotation to a set of *implied* annotations, such that a member that is annotated with the implied annotation gets automatically annotated with each annotation of the implied set.

Consider again the AspectJ specification in Figure 1. This specification contains 6 implications: (i) the first one (lines 3 and 4) specifies that a *PointCut* annotation implies that the component should be ignored by the deadcode checker of Julia (since such methods are just placeholders for the actual pointcut expression), (ii) the other five (lines 5-9) specify that components annotated with *After*, *AfterReturning*, *AfterThrowing*, *Around*, and *Before* should be considered as *EntryPoint*.

2.4 Predicates

A predicate is a construct that lets one assign an arbitrary name to a condition, in order to avoid rewriting it over and over again, and to make the overall specification more readable and maintainable.

Consider again the ASP.NET specification of Figure 2, and in particular lines 4-6. This part of the specification defines a predicate *isNestedComponent* that holds if both field's type and defining class satisfy predicate *isControl*. Such predicate (line 3) holds if a class is subtype of *Control*.

2.5 Specifications

Specifications are the core component of SARL. A specification consists of one or more annotations, and a condition that states when such annotations have to be applied. A specification may target a class, a field, a method, or a method's parameter.

Consider again Figure 2, and in particular lines 17 and 18. These specify to annotate with *ExternallyRead* and *Injected* any field that satisfies predicate *isNestedComponent*.

2.6 Libraries specifications

A framework specification might need to annotate part of the libraries, since the analyzer might need additional annotation on library methods or fields. Therefore, SARL allows one to specify annotations on library components by defining the type of annotations together with the signature of the target program member. Like specifications, a library specification may target a class, a field, a method, or a method's parameter.

Consider now the Windows Form specification in Figure 3, and in particular lines 20, 21 and 22. These specify to annotate method *System.Diagnostics.Process.GetCurrentProcess()* with *ResourceThatDoesNotNeedToBeClosed*, since in fact this method (i) returns a *Disposable* object, but (ii) it is not necessary to close it immediately (as, in fact, this represent the current execution process). Without such annotation, Julia's CloseResource checker would produce a false alarm each time this method is used by the application.

3 Experimental Results

In this section we report our experience when applying SARL to refine the results obtained by Julia on some applications relying on different frameworks (AspectJ [1], ASP.NET, and Windows Forms [5]).

3.1 AspectJ

AspectJ is a Java framework for Aspect Oriented Programming (AOP). In particular, it allows one to annotate methods to identify them as *advices*. An advice is a method that will be automatically invoked when its associated expression (called *pointcut*) is met. Advice annotations specify when the method should be invoked (e.g., @Before states that it should be invoked before the corresponding pointcut). Moreover, AspectJ allows one to annotate methods as *pointcuts*. Such methods become an alias for the actual pointcut expression, allowing their signature to be used instead of the

```
rule: analysis "java"
rule: annotation startsWith "org.aspectj"
implication: "aj.Pointcut" implies
   "SuppressJuliaWarnings(value=deadcode)"
implication: "aj.After" implies "EntryPoint"
implication: "aj.AfterReturning" implies "EntryPoint"
implication: "aj.AfterThrowing" implies "EntryPoint"
implication: "aj.Around" implies "EntryPoint"
implication: "aj.Before" implies "EntryPoint"
```

Figure 1. AspectJ specification, where `aj is a shortcut for org.aspectj.lang.annotation`

actual expression inside an advice. To test this framework, we chose the Apache CXF example [4] from GitHub user *gmateo*. This application is a lightweight example of publishing web services using Apache CXF [3], JAX-WS [8], JAX-RS [7] and SpringFramework [6]. Moreover, it uses a bit of AOP through AspectJ. While Spring, JAX-WS and JAX-RS are natively supported by Julia, AspectJ is not. Hence, such a small application is a good test target for the framework.

Julia's analysis raises 14 warnings. Two of them are deadcode alarms on methods annotated with @Pointcut, thus they are just placeholders for the actual pointcut expression and should not be subjects of such warnings. Figure 1 reports the specification of AspectJ that solves these issues.

The first implication is enough to make the two warnings about pointcuts disappear. However, the same kind of warning would be raised also on advices, since they do not get explicitly called from within the application. Thus, an implication for each available advice is also needed. Note that in our experiment, the only advice present inside the application was not considered deadcode because it is declared as a public method, hence being considered as an entry point for the analysis by Julia.

3.2 ASP.NET

ASP.NET is a Microsoft framework to build C# web applications. The example application for this framework is the one automatically generated by Visual Studio [12] when a new ASP.NET project is created. Such application is chosen since its small dimension makes it easy to fully check the precision of the analysis, hence correctly identifying the components of the specification.

Julia, when analyzing the Visual Studio's ASP.NET generated application, raises 133 warnings: 64 about fields that are never read, 1 about a field that is never written, 4 about useless calls, and 66 about uncalled methods. Most of these warnings are false alarms: every warning about both unread and unwritten fields refers to fields that contain components of the web page, while most of the uncalled methods are event handlers for click events or page loading. Figure 2 reports the specification of ASP.NET framework that solves these issues.

```
rule: superclass "System.Web.HttpApplication"
rule: analysis ".net"
predicate: "isControl" = cls.subtypeOf "System.Web.UI.Control"
predicate: "isNestedComponent" =
   and(fld.type satisfies "isControl",
      fld.definingClass satisfies "isControl")
predicate: "isEventHandler" =
   and(mtd.returnType "void",
      and(mtd.numberOfParameters "2",
         and(mtd.parameter and(par.index "0",
               par.type "System.Object"),
            mtd.parameter and(par.index "1",
               par.type.subtypeOf "System.EventArgs")))))
specification: annotate methods with "EntryPoint"
   if and(mtd.definingClass satisfies "isControl",
         satisfies "isEventHandler")
specification: annotate fields with "ExternallyRead", "Injected"
   if satisfies "isNestedComponent"
```

Figure 2. ASP.NET specification

Using such specification causes the number of warnings to be lowered to 19, composed by 13 true alarms about uncalled methods, 2 true alarms about useless calls, and 4 new true alarms about useless assignments of local variables that were previously not raised since their containing method was wrongly considered deadcode.

3.3 Windows Forms

Windows Forms is a framework to build C# desktop applications. Usually, these UIs are developed through the designer included in Visual Studio, that places each graphical component inside fields, leading to the rise of a high number of warnings about field usage (stating that a field can be replaced by a local variable, or that the value written inside a field is never read later) like ASP.NET. Moreover, each graphical component in Windows Forms implement the IDisposable interface to allow the resources held by such component to be automatically released when the contained form gets closed. As example application for this framework we chose ShareX [10], which is an open-source application for performing screen captures. When analyzing ShareX with Julia, that does not have knowledge of Windows Forms behavior, many warnings about closable objects are raised (stating that a resource is never closed). In fact, out of 5153 warnings, 3222 of them are about closable resources, and 972 are about fields that could be either replaced by local variables, or that hold values never read anywhere in the program. Figure 3 defines the specification of Windows Forms.

This specification lowers the total number of warnings to 1811, with 882 false alarms about closable resources. Such

```
rule: superclass "System.Windows.Forms.Form"
rule: analysis ".net"
predicate: "isComponent" = cls.subtypeOf "System.ComponentModel.
        IComponent"
predicate: "isDisposable" = fld.type.subtypeOf "System.IDisposable"
predicate: "isNestedComponent" =
    and(fld.type satisfies "isComponent",
        fld.definingClass satisfies "isComponent")
specification: annotate fields with "ExternallyRead", "Injected"
    if satisfies "isNestedComponent"
specification: annotate fields with "AutoClosedResource"
    if or(fld.type.subtypeOf "System.Windows.Forms.ContainerControl",
            and(satisfies "isDisposable",
                fld.definingClass satisfies "isComponent"))
library: annotate methods with "
        ResourceThatDoesNotNeedToBeClosed"
    in class "System.Drawing.Brushes"
    if name matches "get_.*()LSystem/Drawing/Brush;"
library: annotate methods with "
        ResourceThatDoesNotNeedToBeClosed"
    in class "System.Drawing.Pens"
    if name matches "get_.*()LSystem/Drawing/Pen;"
library: annotate methods with "
        ResourceThatDoesNotNeedToBeClosed"
    in class "System.Diagnostics.Process"
    if name equals "GetCurrentProcess()LSystem/Diagnostics/Process;"
```

Figure 3. Windows Forms specification

Framework	# alarms	# removed false alarms
AspectJ	14	2
ASP.NET	133	118
Windows Forms	5153	3342

Table 1. Experimental results

false alarms are due to imprecisions of Julia's analyses, and they cannot be addressed by the framework specification.

3.4 Discussion

Table 1 reports the results of the three applications we analyzed and discussed in this section. Despite being very concise (between 9 and 26 lines), the specifications were already effective to remove all the false alarms due to the lack of framework knowledge in the Julia static analyzer. While we expect that when we will apply these specifications to different applications we will need to extend them, we believe that this is already a promising result of the application of SARL to existing software.

4 Conclusion

In this paper, we (i) informally introduced SARL, a domain-specific language to specify the execution model of a framework to instruct a static analyzer, and (ii) applied this approach to three applications dealing with different frameworks, studying how the number of false alarms was reduced thanks to the specification. We are working on the formalization of the language and its extension to other frameworks like Java Lombok [9], .NET Xamarin [14] and Unity [13]. In addition, currently SARL allows one to define a method as an entry point (that is, it might be called with any input state and in any order), but it does not provide any mean to specify the order of execution of such methods. We plan to extend existing annotations in order to support such scenario.

References

[1] AspectJ. 2018. https://www.eclipse.org/aspectj/.

[2] ASP.NET. 2018. https://www.asp.net/.

[3] Apache CXF. 2018. http://cxf.apache.org/.

[4] Apache CXF example. 2018. https://github.com/gmateo/apache-cxf-example/.

[5] Windows Forms. 2018. https://docs.microsoft.com/it-it/dotnet/framework/winforms/.

[6] Spring Framework. 2018. https://spring.io/.

[7] JAX-RS. 2018. https://github.com/jax-rs/.

[8] JAX-WS. 2018. https://github.com/javaee/metro-jax-ws/.

[9] Lombok. 2018. https://projectlombok.org/.

[10] ShareX. 2018. https://getsharex.com/.

[11] F. Spoto. 2016. The Julia Static Analyzer for Java. In *Proceedings of SAS '16 (LNCS)*. Springer.

[12] Visual Studio. 2018. https://www.visualstudio.com/.

[13] Unity. 2018. https://unity3d.com/.

[14] Xamarin. 2018. https://visualstudio.microsoft.com/xamarin/.