SARL: OO Framework Specification for Static Analysis

Pietro Ferrara[1] and Luca Negrini[2,1]

[1] Ca' Foscari University of Venice, Italy pietro.ferrara@unive.it
[2] JuliaSoft SRL, Verona, Italy luca.negrini@juliasoft.com

Abstract. Semantic static analysis allows sound verification of program properties, that is, to prove that a given property holds for all possible executions. However, modern object-oriented applications make heavy use of third-party *frameworks*. These provide various functionalities (like libraries), as well as an extension of the execution model of the program. Applying standard models to statically analyze software relying on such frameworks could be potentially unsound and imprecise.
In this paper we introduce SARL, a domain-specific language which allows one to specify the runtime behaviors of frameworks of object-oriented programs. Such specifications can be then applied to automatically generate annotations on program components of the application to model the framework runtime environment. In addition, SARL specifications document which aspects of a framework are supported by the static analyzer and how. We adopted SARL to model WindowsForms and ASP.NET, two of the most popular .NET frameworks in an existing industrial static analyzer (Julia). We then analyzed the three most popular GitHub repositories using these frameworks, comparing the results with and without SARL. Our experimental results show that the application of SARL sensibly improved the precision and soundness of the analysis without affecting its runtime performances.

1 Introduction

Static analysis allows one to prove properties of computer programs without executing them. Such properties vary from the absence of runtime errors to functional correctness.

Object-oriented software makes extensive use of third party libraries and frameworks, avoiding the re-implementation of common functionalities in favor of reusable and highly tested code already widely used. In object-oriented software, a library is a collection of classes, methods, etc. that implements some standard functionalities, and that can be called by the application. Instead, software frameworks represent a wider concept: *"A software framework provides a standard way to build and deploy applications. (...) Software frameworks may include support programs, compilers, code libraries, tool sets, and application programming interfaces (APIs) which bring together all the different components to enable development of a project or system."*[25]. Frameworks have been applied to various contexts. For instance, ASP.NET [9] allows a developer to implement and deploy a web application, while WindowsForms [14] is designed to easily build desktop applications with modern UIs. Therefore, each framework provides a specific execution model.

Static analyzers may raise alarms on generated code or specific framework components, since these usually follow non-standard patterns that are hardly detectable by and might confuse the analyzers. Moreover, frameworks often offer ad-hoc execution models, that might rely on specific configuration files. Not being aware of these behaviors might lead to unsoundly ignoring relevant portions of the application code, or to consider too many methods as candidates for the reflective calls to preserve soundness.

Nowadays, there are dozens of software frameworks (like Spring and Lombok in Java, or ASP.NET and WindowsForms in C#), each one with its own execution model, and new ones keep emerging. This represents a challenge for static analyzers, since customers expect the analysis to be up-to-date with modern technologies, while the effort of modeling even a single framework might not be negligible. Furthermore, to keep amplifying the range of supported frameworks, there is the need for a flexible mean to model new ones, as well as to improve the knowledge of the analyzer on already-known frameworks. Finally, new versions of the same framework might require different models. It is therefore essential to document which parts of a framework are supported and how to keep the models updated when new versions are released.

Contribution In this paper we present `SARL` (Static Analysis Refining Language), a domain-specific language which allows one to easily instruct a static analyzer about the execution model of a framework to *improve* the results in terms of both precision and soundness. `SARL` targets statically type-safe, object-oriented programming languages (C# and Java in particular). These languages offer a construct which allows to add metadata to object-oriented components, describing their characteristics. We will generically refer to it as *annotation*.

`SARL` adopts annotations as the key mean to instruct the analyzer about both the structure and the execution model of the application under analysis. The goal of `SARL` is to produce a set of rules, called *framework specification*, that describes the behavior of a framework. Such description is then automatically applied to a program to produce a collection of annotations on it that the static analyzer is able to interpret and exploit during the analysis (e.g., when building the call graph of the program or approximating the heap structure).

Since we want to apply the `SARL` specification and extract the annotations on the target application *before* the analysis starts, framework specifications must be evaluated syntactically on the analyzed application without any semantic knowledge of the program. At this stage, no information about the dynamic types of the program's values is known, as well as the effective targets of methods calls. Hence, the content of framework specifications needs to be designed using static types and call targets.

`SARL` has been interfaced with Julia, an abstract interpretation based static analyzer of Java bytecode whose analyses consider a wide range of annotations.

Born as a Java analyzer, Julia has been recently extended to analyze .NET (CIL) bytecode as well [13].

Therefore, we applied `SARL` to model two popular .NET frameworks (WindowsForms and ASP.NET), and we present, for each of these frameworks, the results of Julia analyses on the 3 most popular GitHub repositories of projects relying on these frameworks. In particular, we study how the analysis improved in terms of precision and soundness when using the `SARL` specification w.r.t. the original Julia analysis. The experimental results show that for programs widely relying on a framework (WindowsForms) the improvement is dramatic (`SARL` specification removed between 35.3% and 74.5% of false alarms), while when only a small portion of a program exploits a framework (ASP.NET) the benefit is restricted to that (between 1.6% to 4.3%). Moreover, since `SARL` specifications collect all framework support into a single file, a readable representation of what has been covered and how can be easily generated. Julia includes this representation as part of its own documentation[3].

The rest of the paper is structured as follows. Section 1.1 reports a first example of a `SARL` specification modeling ASP.NET. Section 2 discusses the related work, while Section 3 reports the overall architecture of the Julia static analyzer. Section 4 introduces another specification, targeting WindowsForms, and uses it to describe all the constructs of `SARL`, together with its complete grammar. Section 5 reports the benefits obtained when applying `SARL` specifications to six applications using WindowsForms and ASP.NET, while Section 6 concludes.

1.1 Example `SARL` specification

Consider the specification for the ASP.NET [9] framework contained in Figure 1, where namespaces have been omitted for the sake of compactness. ASP.NET is a Microsoft framework used to build web applications written in C#. It comes in various flavors, like *WebForms* and *MVC*. As most web application frameworks, applications written with ASP.NET have an execution model fairly different from the one of a standard application. In fact, a wide variety of methods are invoked from the external environment, such as page and graphical event handlers. This means that a static analyzer would not find explicit calls to these methods, considering them (as well as all other methods invoked directly or indirectly) as not reachable. Moreover, graphical objects are usually stored in fields, enabling the runtime environment to access them for initialization. Since those interactions are not part of the program, a static analyzer might suggest to replace those fields with local variables (if it is referenced in only one method), or to remove them completely (if no explicit accesses are found in the code). The specification of Figure 1 describes both these features (as well as marking few other methods as reachable), providing metadata (as annotations supported by Julia), in a concise and self-explanatory manner: it is applied when a .NET program contains at least one *HttpApplication*, which is the base class for ASP.NET applications (lines 1 and 2). Then, fields representing runtime-managed objects

[3] `https://static.juliasoft.com/docs/2.7.0.3/frameworks.html`

```
1  rule: rte ".net"
2  rule: superclass HttpApplication
3  predicate: isControl = cls -> subtypeOf:: "Control"
4  predicate: isNestedComponent = and(fld -> type satisfies isControl, fld ->
       definingClass satisfies isControl)
5  predicate: isEventHandler = and(mtd -> basicReturnType:: "void", and(mtd ->
       numberOfParameters:: 2, and(mtd -> hasParameter and(par -> index:: 0, par
       -> type:: "Object"), mtd -> parameter and(par -> index:: 1, par -> type.
       subtypeOf:: "EventArgs")))))
6  predicate: isWebViewExecute = and(mtd -> basicReturnType:: "void", mtd -> name
       :: "Execute")
7  predicate: isGetAppInstance = and(mtd -> returnType.subtypeOf:: "HttpApplication",
       mtd -> name:: "get_ApplicationInstance")
8  specification: annotate mtd with EntryPoint if and(mtd -> definingClass satisfies
       isControl, satisfies isEventHandler)
9  specification: annotate fld with ExternallyRead, Injected if satisfies isNestedComponent
10 specification: annotate mtd with EntryPoint if and(mtd -> definingClass.subtypeOf::
       "WebPageExecutingBase", and(mtd -> numberOfParameters:: 0, or(satisfies
       isWebViewExecute, satisfies isGetAppInstance)))
11 specification: annotate mtd with EntryPoint if and(mtd -> definingClass::startsWith "
       __ASP.FastObjectFactory", and(mtd -> name::startsWith "Create_ASP_", and(
       mtd -> returnType:: "System.Object", mtd -> numberOfParameters:: 0)))
```

Fig. 1. ASP.NET specification

(subtypes of the *Control* class) are identified and marked (line 9) as externally read and written (that is, injected), while event handlers, web page creation factories, and other standard framework methods are considered as entry point (lines 10-11).

This information is needed in order to build up a sound approximation of the execution of ASP.NET applications. For instance, event handlers are usually not public, but they are implicitly executed by the framework. Therefore, these methods would be considered as unreachable by a semantic static analyzer, potentially leading to both false positives (e.g., warnings about unreachable event handlers) and negatives (e.g., missing warnings on the code implementing of the event handler). Instead, by annotating them as entry points, SARL instructs the analyzer (and the call graph constructor in particular) to consider them as externally called with arbitrary values. In this way, the analysis will produce alarms on the code directly or indirectly executed by the method (removing false negatives), and remove warnings about unreachable event handlers (removing false positives). Similarly, ASP.NET stores objects representing UI components in (usually private) fields that are written and read by the framework itself. Usually such fields are never written in the application, and sometimes not even read. For instance, in the first case, a semantic static analyzer would consider these fields as always null, potentially producing both false positives (e.g., nullness alarms when the object stored in the field is dereferenced) and negatives (e.g., on the code of a branch of an if-then-else statement that is guarded by a nullness check on the field, thus considered as deadcode). These imprecisions of the analysis are removed once these fields are annotated as externally written (injected), since the analyzer will consider that they might be assigned with arbitrary values.

As one can see from this brief example, `SARL` allows to easily specify to which programs the specification should be applied, a set of predicates (improving the readability and reusability of the specification), and a set of specification rules to annotate program components as well as libraries. This specification will later be used in Section 5 to refine the results of Julia on applications that use ASP.NET.

2 Related work

Several static analyzers, like Julia and FindBugs [15] allow a developer to instruct the analysis about specific runtime behaviors of a framework through annotations. The goal of `SARL` is to automatically produce these annotations and apply them to the code under analysis. In this way, `SARL` decouples the framework specification from the program.

Specification languages such as the Java Modeling Language [17] allow to specify pre- and post-conditions and object invariants following the design-by-contract methodology. Different verification tools can then check if the program satisfies the given specification. Therefore, these languages are aimed at specifying the properties of interests that one might want to check on a program, rather than the behavior of frameworks (that is the goal of `SARL`).

SLIC is a specification language developed about two decades ago "designed to specify the temporal safety properties of APIs implemented in the C programming language" [10]. Similarly to `SARL`, SLIC was designed to specify the behavior of libraries, and in particular safety temporal requirements of the APIs. Instead, `SARL` is focused towards object-oriented frameworks that might both provide external libraries and modify the runtime execution model of the program, and it targets various safety and security properties of such programs.

Previous works [11,23] relied on hardcoded knowledge of specific framework features, hence building an a priori model for each framework. However, handling new frameworks required a modification of the analysis engine, expanding both the size and complexity of the product and requiring a developer with expertise both in the framework itself and on static analysis. Moreover, this solution did not provide a fast and reliable way for supporting new software frameworks, and it did not document which features of a specific framework are taken into account by the analysis and how.

More recent works exploited a framework's configuration files. These files often restrict the possible executions, allowing to (almost always) precisely resolve the targets of reflective calls in the framework. F4F [22] aimed at building an application-specific model of the framework's behavior automatically, and this can be used by the analysis to react accordingly to modifications of the execution model made by the framework itself. However, building a model generator for each framework does not keep the actual pace of releases of new frameworks, and each analysis needs to be modified to be model-aware during its execution.

Concerto [16] combined mostly-concrete interpretations of the framework code and abstract interpretation of application code, providing sound and accurate analysis on the overall program. Both of the above approaches targeted frameworks whose behavior depends on some application-specific configuration files, and that is not the case for any framework. Moreover, the code from the framework itself needs to be submitted to the analysis, and this will eventually slow down the analyzer due to the significantly larger amount of code to analyze. Also notice that, if the format of the configuration file changes among different versions of the same framework, a new parser for such a file must be built, and the logic of the newly introduced constructs must be embedded into the analyzer.

StubDroid [8] built up data flow summaries of Android libraries for taint analyzers. If on the one hand such approach is completely automatic, on the other hand it is specific for taint analysis and it required an ad-hoc static analysis in order to infer the data flow summaries. While `SARL` framework specifications might be automatically inferred with static and dynamic analyses, this is not the focus of this paper and is left as future work. Averroes [7] introduced a new approach that, starting from the code of application and libraries, built up a placeholder library that soundly approximates the library behaviors. The construction of the placeholder library relied on the separate compilation assumption, and it handles reflection. Such an approach sensibly improved the efficiency of the analysis without affecting its precision and soundness. However, more recent frameworks rely on ad-hoc runtime environments that extends the execution model of the programming languages. These runtime environments are outside the code of the library, and therefore they cannot be handled with this approach.

3 Julia

Julia's analyses are interprocedural (that is, they consider the flow of control and information from callers to callees and vice-versa) and abstract the heap (that is, they consider the flow of data through heap writes and reads). This is essential to perform semantic static analyses, such as information flow or sound nullness analysis.

In Julia, the model of the program under analysis is built by a so-called class analysis, that infers the possible runtime dynamic types of the variables and stack elements. Julia uses the one defined in [19], which has been shown to be a reasonable trade-off between precision and cost. The construction of such model of the program is called *extraction* in Julia, since methods are extracted and then analyzed only if they are actually called in the program. The extraction starts from a set of entry points, that, by default, are all the public methods of the analyzed application. However, as an input of the analysis the user can specify other entry point modes, and in particular (i) only standard entry point methods (e.g., *main* and servlet methods), (ii) only explicit entries (that is, methods annotated as *@EntryPoint*), or (iii) all public and protected methods. For the sake of simplicity, in the rest of the paper we consider only the default mode. Julia includes various static analyses (e.g., sound nullness analysis [20], taint analysis [12] and data-size analysis [21]).

Being born as a Java analyzer, Julia acquired knowledge on widespread Java frameworks during the years. However, frameworks behaviors have been hardcoded throughout various analysis components, making it hard to understand and document which aspects of each framework has been covered and how. Instead, Julia has no hardcoded model for C# frameworks. Thus, our initial effort targets this area.

Annotations As of version 2.7.0.3, Julia defines more than 70 annotations with various meanings[4]. Some of them are used to provide context about how the application interacts with the external environment (e.g., *@EntryPoint* states that a method could be called from outside the program, while *@Injected* states that a field or a parameter could be written by an external source), while the majority of them are used to provide information to a specific checker (e.g., *@SqlTrusted* is used to instruct the Injection checker that untrusted data should not flow into that location since it will end up in a database, while *@NonNull* states that a field or a method's return value are never *null*). Finally, *@SuppressJuliaWarnings* instructs Julia that a certain kind of warning should not be reported on the annotated component (either a field, method, constructor, class, method parameter, or local variable).

Each annotation has a different scope, and thus, a different impact on the analysis: *@EntryPoint* will be exploited during the construction of the call graph, but its effect will be propagated throughout the whole analysis (i.e., additional reachable code will be considered); *@SqlTrusted* instead will only be used during the execution of the taint analysis based checkers (the Injection checker is the most popular, but other ones exist). Thus, a *framework specification* could be logically split into sections, each one having effects on a different set of checkers.

4 The `SARL` Language

The goal of `SARL` is to allow a user to specify a set of rules (called *framework specification*) representing how the framework affects the runtime behavior of a program. Such specification will then be exploited by a static analyzer to improve its precision and soundness. In particular, we rely on annotations to pass this information. Therefore, these should be expressive enough to represent these runtime behaviors. Throughout this paper, we need annotations to specify: (i) when a method might be called by the framework runtime (*@EntryPoint* in Julia), (ii) when a field is read or written by the framework runtime (*@ExternallyRead* and *@Injected* in Julia), (iii) when the warnings on a specific component (e.g., method or field) should be suppressed (*@SuppressJuliaWarnings* in Julia, *@SuppressWarnings* in Java), and (iv) properties related to specific analyses (e.g., *@AutoClosedResource* of the CloseResource analysis in Julia).

[4] The documentation of the available annotations is available at `https://static.juliasoft.com/docs/2.7.0.3/annotations.html`

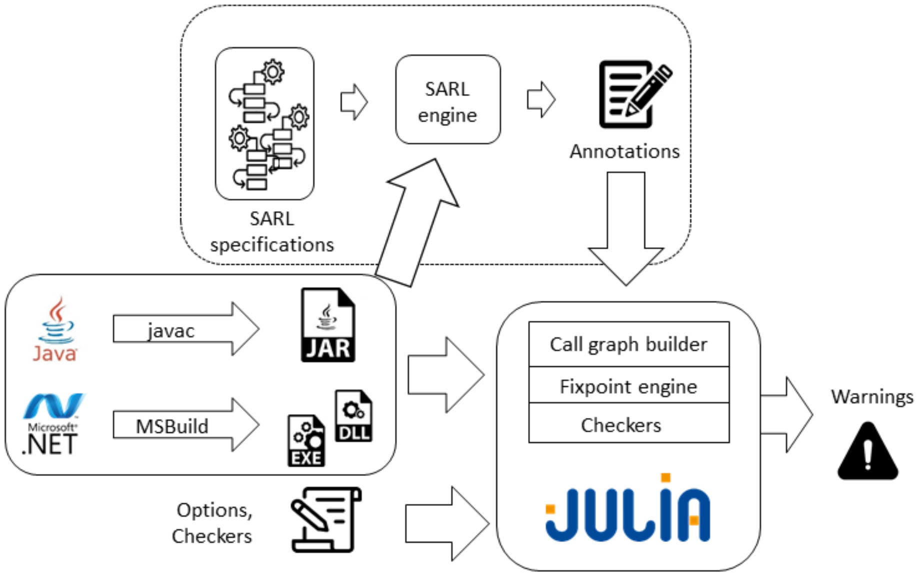

Fig. 2. Schema of Julia's architecture with `SARL`

Building a *framework specification* can be achieved with two different approaches. One can acquire knowledge about the framework itself, understanding its model of execution and how it interacts with the application code. Then, the acquired knowledge needs to be converted into a `SARL` specification, by understanding how each framework feature may impact the various analysis modules. While this approach ensures that every peculiarity of the framework has been taken into account, the number and heterogeneity of software frameworks makes it hard to achieve, since one should possess knowledge on both the analyzer *and* the framework. Another approach consists into iteratively analyzing software that rely on the target framework, inspecting the analysis results searching for evidence of the lack of framework knowledge by the analyzer (e.g., unreachable methods that are instead invoked by the framework, no injection-related warning on a web application, . . .) and fixing them in the specification. This approach is highly dependent on how representative the software is in exploiting the framework's functionalities, but can nevertheless be a good starting point. Both `SARL` instances presented in this paper have been built following the latter approach.

Figure 2 depicts the overall architecture of our approach, and how this interfaces with the Julia static analyzer. Given a framework specification and an application to be analyzed, the `SARL` engine (represented inside the dotted rectangle) produces a set of annotations. These are then serialized to an XML file and passed together with all the other inputs of the analysis (analyzed code, analysis options and checkers to run) to the analyzer. Further developments will bring the `SARL` engine inside the analyzer itself, making the identification of

```
rule: rte ".net"
rule: superclass Form
predicate: isComponent = cls −> subtypeOf:: "IComponent"
predicate: isDisposable = fld −> type.subtypeOf:: "IDisposable"
predicate: isNestedComponent = and(fld −> type satisfies isComponent, fld −>
    definingClass satisfies isComponent)
predicate: isGeneratedFormField = and(fld −> definingClass.subtypeOf:: "
    ContainerControl", and(fld −> hasAccessor:: "private", and(fld −> name:: "
    components", fld −> type.subtypeOf:: "IContainer")))
specification: annotate fld with ExternallyRead, Injected if satisfies isNestedComponent
specification: annotate fld with AutoClosedResource if or(fld −> type.subtypeOf:: "
    ContainerControl", and(satisfies isDisposable, fld −> definingClass satisfies
    isComponent))
specification: annotate fld with NonNull, ExternallyRead, Injected if satisfies "
    isGeneratedFormField";
library: annotate mtd with ResourceThatDoesNotNeedToBeClosed if cls Brushes mtd
    matches "get_.*()LSystem/Drawing/Brush;"
library: annotate mtd with ResourceThatDoesNotNeedToBeClosed if cls Pens mtd
    matches "get_.*()LSystem/Drawing/Pen;"
library: annotate mtd with ResourceThatDoesNotNeedToBeClosed if cls Process mtd "
    GetCurrentProcess()LSystem/Diagnostics/Process;"
```

Fig. 3. WindowsForms specification

frameworks and the generation of extra annotations a fixed step of the analysis lifecycle.

Running example: Windows Forms Before discussing SARL formally, we introduce another SARL specification targeting WindowsForms [14], a framework to build GUIs of C# desktop applications. Usually, these are developed through the designer included in Visual Studio, which places a pointer to each graphical component in private fields initialized by the generated code, causing the analyzer to raise 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). Moreover, each graphical component in WindowsForms implements the *IDisposable* interface (which represents objects that should be disposed when no longer needed, since they could hold handles to non-managed resources that needs to be manually released) and it is disposed by the framework runtime. Figure 3 reports the specification of WindowsForms (as in Section 1.1, namespaces have been omitted for compactness), that will be used to explain SARL constructs.

Language Definition SARL is built over five basic components: *rules*, *implications*, *specifications*, *predicates*, and *library specifications*. Rules embed information to detect if a given application relies on the framework. The specification is applied if and only if the condition specified in the rules holds. Core components (implications, specifications and predicates) allow one to define the conditions required to apply an annotation to a program member. Finally, library speci-

```
                 General structure
      ⟨SARL⟩::=⟨RULES⟩
                   (⟨IMPL⟩
                   | ⟨PREDICATE⟩
                   | ⟨SPEC⟩
                   | ⟨LIB⟩)*
     ⟨RULES⟩::=(rule: (⟨RULE_RTE⟩ |
                     ⟨RULE_CODE⟩))*
  ⟨RULE_RTE⟩::=⟨R_RTE⟩
 ⟨RULE_CODE⟩::=⟨R_ANN⟩
                   | ⟨R_SUPER⟩
                   | ⟨R_TYPE⟩
      ⟨IMPL⟩::=implication: ⟨ANN⟩
               implies ⟨ANN⟩ (,⟨ANN⟩)*
 ⟨PREDICATE⟩::=predicate: ⟨ID⟩ = ⟨C_CHAIN⟩
      ⟨SPEC⟩::=specification: annotate ⟨T⟩ with ⟨ANN⟩
                   (, ⟨ANN⟩)* if ⟨C_CHAIN⟩
       ⟨LIB⟩::=library: annotate ⟨T⟩
                   with ⟨ANN⟩ (,⟨ANN⟩)*
                   if ⟨SIG⟩
                   Rules
     ⟨R_RTE⟩::=rte (⟨R_OP⟩)? ⟨STRING⟩
     ⟨R_ANN⟩::=annotation (⟨R_OP⟩)? ⟨QNAME⟩
   ⟨R_SUPER⟩::=superclass (⟨R_OP⟩)? ⟨QNAME⟩
    ⟨R_TYPE⟩::=uses type (⟨R_OP⟩)? ⟨QNAME⟩
                 Conditions
   ⟨C_CHAIN⟩::=⟨NT_TYPE⟩ ->
            ⟨NT_OP⟩(⟨OPT⟩)?.⟨NT_COND⟩
    | ⟨T_TYPE⟩ -> ⟨T_OP⟩ ⟨VALUE⟩ | ⟨L_COND⟩
    ⟨L_COND⟩::=| satisfies ⟨ID⟩ | not(⟨C_CHAIN⟩)
                   | and(⟨C_CHAIN⟩,⟨C_CHAIN⟩)
                   | or(⟨C_CHAIN⟩,⟨C_CHAIN⟩)

  ⟨NT_COND⟩::=⟨NT_OP⟩ . ⟨NT_COND⟩
                  | ⟨NT_OP⟩⟨OPT⟩::⟨T_COND⟩
                  | (⟨NT_OP⟩)?::⟨T_COND⟩
                  | ⟨NT_OP⟩⟨L_COND⟩
   ⟨T_COND⟩::=(⟨T_OP⟩)?⟨VALUE⟩
      ⟨OPT⟩::=[⟨ID⟩ ]
                   Targets
        ⟨T⟩::=cls | fld | mtd | par
      ⟨SIG⟩::=cls ⟨QNAME⟩ (fld ⟨T_OP⟩ ⟨STRING⟩
                  | mtd ⟨T_OP⟩ ⟨STRING⟩
                      (par ⟨NUMBER⟩)?)?
                    Types
  ⟨NT_TYPE⟩::=cls | ann | par | var | mtd | fld
   ⟨T_TYPE⟩::=str | int
                  Operators
     ⟨R_OP⟩::=equals | startsWith | endsWith | contains
     ⟨T_OP⟩::=equals | startsWith | endsWith
              | contains | matches
    ⟨NT_OP⟩::=definingMethod | name | index | type
                  | basicType | hasAnnotation
                  | definingClass | returnType
                  | basicReturnType | hasVariable
                  | hasAccessor | hasOptionValue
                  | hasParameter | subtypeOf
                  | containsMethod | containsField
                  | numberOfParameters
         Program members identifiers
      ⟨ANN⟩::=⟨QNAME⟩ ((⟨MEMBER⟩
                  (, ⟨MEMBER⟩)*))?
   ⟨MEMBER⟩::=⟨ID⟩ = ⟨STRING⟩
    ⟨QNAME⟩::=⟨ID⟩ (. ⟨ID⟩)*
       ⟨ID⟩::=[a-zA-Z] | [_a-zA-Z]([_a-zA-Z0-9])*
                   Values
    ⟨VALUE⟩::=⟨STRING⟩ | ⟨NUMBER⟩
   ⟨STRING⟩::=“ .* ”
   ⟨NUMBER⟩::=0 | [1-9]([0-9])*
```

Fig. 4. `SARL`'s grammar

fications allow to generate annotations also on non-application classes (that is, classes that come from a supporting library or the system runtime).

Figure 4 defines the complete syntax of `SARL`, while Figure 5 formalizes the semantics of the various components. During the formalization, we consider a program p as a set of classes; each class is a tuple (n, A, F, C) where n is the name of the class, while A, M and F are the set of annotations, fields and methods of the class, respectively. An annotation is a tuple $(n, \wp(n \times str))$, with n being a full qualified name and $\wp(n \times str)$ being the set of members, represented as a pair of name and string value. A field is a tuple $(n, t, A, \wp(str))$, where n is the name, t is the type, A is the set of annotations, and $\wp(str)$ is the set of accessors. A method is a tuple $(n, t, A, \wp(str), P, V)$, where n is the name, t is the return type, A is the set of annotations, $\wp(str)$ is the set of accessors, P is the set of parameters, and V is the set of local variables. A parameter is a tuple (n, t, A, i), with n being the name, t being the type, A being the set of annotations, and i being the index of the parameter. A variable instead is a pair (n, t) with n being the name and t being the type. Each element of the formalization could be subscripted with a letter stating if it refers to a class c, a field f, a method m, or a parameter p.

The semantics, that will be discussed in the rest of this section, relies on a set of standard operators over the different object-oriented program components informally defined in Tables 6 and 7 in Appendix A. For the sake of simplicity, from now on we denote with ⟨XS⟩ sequences or sets of ⟨X⟩ components.

Rules A rule ⟨RULE⟩ defines a condition to be satisfied to apply a specification. Rules express conditions on either the analysis ⟨RULES_RTE⟩, or the code ⟨RULES_CODE⟩. Rules semantics is defined in the first five definitions of Fig-

$$hold((\langle \text{RULES\_RTE}\rangle, \langle \text{RULES\_CODE}\rangle), \mathtt{p}) \Leftrightarrow \begin{cases} \langle \text{RULES\_RTE}\rangle = \emptyset \vee \exists r \in \langle \text{RULES\_RTE}\rangle : hold(r, \mathtt{p}) \\ \wedge \\ \langle \text{RULES\_CODE}\rangle = \emptyset \vee \exists r \in \langle \text{RULES\_CODE}\rangle : hold(r, \mathtt{p}) \end{cases}$$
$$hold(\langle \text{R\_RTE}\rangle, \mathtt{p}) \Leftrightarrow holdString(extractRTE(\mathtt{p})\ \langle \text{R\_OP}\rangle\ \langle \text{STRING}\rangle))$$
$$hold(\langle \text{R\_ANN}\rangle, \mathtt{p}) \Leftrightarrow \exists \mathtt{n} \in extractAnn(\mathtt{p}) : holdString(\mathtt{n}\ \langle \text{R\_OP}\rangle\ \langle \text{QNAME}\rangle))$$
$$hold(\langle \text{R\_SUPER}\rangle, \mathtt{p}) \Leftrightarrow \exists \mathtt{n} \in \mathtt{p} : \mathtt{n}' \in extOrImpl(\mathtt{n}) : holdString(\mathtt{n}'\ \langle \text{R\_OP}\rangle\ \langle \text{QNAME}\rangle))$$
$$hold(\langle \text{R\_TYPE}\rangle, \mathtt{p}) \Leftrightarrow \exists \mathtt{n} \in extractType(\mathtt{p}) : holdString(\mathtt{n}\ \langle \text{R\_OP}\rangle\ \langle \text{QNAME}\rangle))$$

$$impl(\langle \text{IMPL}\rangle, \mathtt{p}) = \bigcup_{(\mathtt{n_c}, \mathtt{A_c}, \mathtt{F}, \mathtt{M}) \in \mathtt{p}} \left\{ \begin{array}{l} (n_c, A_c \cup ann(A_c, \langle \text{IMPL}\rangle), F', M') : \\ F' = \bigcup\limits_{f \in F} (n_f, t_f, A_f \cup ann(A_f, \langle \text{IMPL}\rangle), C_f), \\ M' = \bigcup\limits_{m \in M} \left\{ \begin{array}{l} (n_m, t_m, A_m \cup ann(A_m, \langle \text{IMPL}\rangle), C_m, P'_m, V_m) : \\ P'_m = \bigcup\limits_{p \in P_m} (n_p, t_p, A_p \cup ann(A_p, \langle \text{IMPL}\rangle), i_p)\} \end{array} \right\} \end{array} \right\}$$
$$ann(A, \langle \text{ANN}\rangle\ implies\ \langle \text{ANNS}\rangle) = A \cup \begin{cases} \emptyset & \text{if } \langle \text{ANN}\rangle \notin A \\ \langle \text{ANNS}\rangle & \text{if } \langle \text{ANN}\rangle \in A \end{cases}$$

$$lib(\langle \text{LIB}\rangle, \mathtt{p}) = \bigcup_{(\mathtt{n_c}, \mathtt{A_c}, \mathtt{F}, \mathtt{M}) \in \mathtt{p}} \left\{ \begin{array}{l} (n_c, A_c \cup addLA(\langle \text{LIB}\rangle, n_c), F', M') : \\ F' = \bigcup\limits_{f \in F} (n_f, t_f, A_f \cup addLA(\langle \text{LIB}\rangle, f), C_f), \\ M' = \bigcup\limits_{m \in M} \left\{ \begin{array}{l} (n_m, t_m, A_m \cup addLA(\langle \text{LIB}\rangle, m), C_m, P'_m, V_m) : \\ P'_m = \bigcup\limits_{p \in P_m} (n_p, t_p, A_p \cup addLA(\langle \text{LIB}\rangle, p), i_p)\} \end{array} \right\} \end{array} \right\}$$
$$addLA(\langle \text{LIB}\rangle, s) = \begin{cases} \langle \text{ANNS}\rangle & \text{if } typeOf(s) = \langle \text{T}\rangle \wedge checkSignature(s, \langle \text{SIG}\rangle) \\ \emptyset & \text{otherwise} \end{cases}$$

$$spec(\langle \text{SPEC}\rangle, \mathtt{p}) = \bigcup_{(\mathtt{n_c}, \mathtt{A_c}, \mathtt{F}, \mathtt{M}) \in \mathtt{p}} \left\{ \begin{array}{l} (n_c, A_c \cup cond(\langle \text{SPEC}\rangle, (n_c, A_c, F, M)), F', M') : \\ F' = \bigcup\limits_{f \in F} (n_f, t_f, A_f \cup cond(\langle \text{SPEC}\rangle, f), C_f), \\ M' = \bigcup\limits_{m \in M} \left\{ \begin{array}{l} (n_m, t_m, A_m \cup cond(\langle \text{SPEC}\rangle, m), C_m, P'_m, V_m) : \\ P'_m = \bigcup\limits_{p \in P_m} (n_p, t_p, A_p \cup cond(\langle \text{SPEC}\rangle, p), i_p)\} \end{array} \right\} \end{array} \right\}$$
$$cond(\langle \text{SPEC}\rangle, s) = \begin{cases} \langle \text{ANNS}\rangle & \text{if } typeOf(s) = \langle \text{T}\rangle \wedge chain(s, \langle \text{C\_CHAIN}\rangle) \\ \emptyset & \text{otherwise} \end{cases}$$

Fig. 5. Semantics of `SARL` statements, where $< i >_k$ represents element i of k (e.g., n_m represents the name of method $m \in M$)

ure 5. Analysis rule ⟨R_RTE⟩ defines the runtime environment (e.g., .NET or Java) of the framework.Instead, code rules define what should be found inside the application to apply a specification, in particular identifying some specific types (either as supertype - ⟨R_SUPER⟩, or as type in a member signature - ⟨R_TYPE⟩), or annotations from the library (⟨R_ANN⟩).

Example The first two lines of the specification of WindowsForms in Figure 3 define the ⟨RULES⟩. In particular, this specifies to apply the framework to applications whose (i) runtime environment is set to *.net* (line 1), and (ii) at least one class inherits from (or implements) *Form* class (line 2).

Implications ⟨IMPL⟩ specifies that a given annotation ⟨ANN⟩ implies a set of other annotations. Then, if a program member is annotated with the first annotation, it is automatically annotated with all the other annotations (definitions of *impl* and *ann* in Figure 5). This can be useful in situations where the developers have used annotations from the libraries to get some functionalities in their code, and these annotations semantically imply some other annotations supported by the analyzer, or when a framework searches a program member through reflection by searching all annotated members. For example, if a Java method is annotated with JAX-RS's *javax.ws.rs.GET* annotation, it will eventually be called from the external environment to handle an HTTP GET request. Hence, such method has to be considered an entry point of the analysis. Thus, relatively to Julia, an implication between *@GET* and *@EntryPoint* is needed.

Predicates ⟨PREDICATE⟩ lets one assign an arbitrary name ⟨ID⟩ to a condition ⟨C_CHAIN⟩ (later defined in this Section), in order to avoid rewriting it multiple times. For example, one might define predicate *isGetter* whose condition identifies a getter method. Once such predicate is defined, its name can be used in any other condition instead of rewriting the actual condition.

Example For instance, line 5 of the WindowsForms specification in Figure 3 defines the *isNestedComponent* predicate. This holds if and only if the type of the given field satisfies predicate *isComponent* (that is, it is a subtype of *IComponent* as defined at line 3), and the class defining the field satisfies *isComponent* as well (that is, it is a subtype of *IComponent*).

Specifications This is the core component of `SARL`. A specification lets one specify a condition on a program member that, when satisfied, causes a set of annotations to be generated on that program member. Thus, all such members have to be iterated when evaluating a specification. This construct enables one to identify members depending on their structure, as well as the one of their related members. This goes beyond the simple reflective access (e.g., the one offered by library specifications described below), allowing one to identify members in a very precise manner. ⟨SPEC⟩ consists of the type ⟨T⟩ of program member we want to annotate, one or more annotations ⟨ANN⟩, and a condition ⟨C_CHAIN⟩ that states when these have to be applied (definition of *spec* in Figure 5, where *typeOf* returns the type - class, field, method, or parameter - of a program component). For example, when analyzing a Unity [24] application, each *Start* method of classes that extends *UnityEngine.MonoBehaviour* should be considered as an entry point for the analysis, since such method will be called by the Unity engine to perform the setup of the component. This can be achieved using a specification that has *mtd* as target, contains *@EntryPoint* as annotation, and as condition the *and* of the two aforementioned conditions (method's name and parent class).

Example For instance, line 9 of WindowsForms specification (Figure 3) specifies to annotates all fields satisfying *isNestedComponent* field with *@ExternallyRead* and *@Injected*.

Conditions ⟨C_CHAIN⟩ may be (i) the application of a predicate (via its name), (ii) a logical operator (*and*, *or*, *not*) applied to other conditions, (iii) a non-terminal operator followed by a further condition, or (iv) a terminal operator followed by a constant value where *str* and *int* represent strings and integers, respectively. Conditions are grouped in chains, where operators are applied to navigate among the properties of program members (e.g., starting from a class, one could navigate to a parameter of one of its supertype's methods). The ability to navigate through program members enables the definition of syntactic conditions that corresponds to how a framework might search for a program member to in-

```
function applySARL((a, l), ⟨SARL⟩)
    rte ← ⟨RULES_RTE⟩ ∈ ⟨RULES⟩ ∈ ⟨SARL⟩
    code ← ⟨RULES_CODE⟩ ∈ ⟨RULES⟩ ∈ ⟨SARL⟩
    if $hold((rte, code), a)$ then
        for ⟨SPEC⟩ ∈ ⟨SARL⟩ do
            a ← spec(⟨SPEC⟩, a)
        for ⟨LIB⟩ ∈ ⟨SARL⟩ do
            a ← lib(⟨LIB⟩, a)
            l ← lib(⟨LIB⟩, l)
        for ⟨IMPL⟩ ∈ ⟨SARL⟩ do
            a ← impl(⟨IMPL⟩, a)
            l ← impl(⟨IMPL⟩, l)
    return (a, l)
```

Fig. 6. Application of a `SARL` specification to a program

teract, both by searching instances of particular types or by retrieving members annotated with a given framework annotation. The formalization of the check of these conditions is represented by function *chain* in Figure 5 and left implicit for the sake of simplicity (mostly standard checks of standard OO properties). Notice that, if one omits an operator, the default one will be applied, depending on the program member that is currently under evaluation (recall Table 6 from Appendix A).

Library specifications In object-oriented software, most of the code is contained in libraries providing standard features to the application. However, libraries contain code which could need `SARL` generated annotations, since their methods or fields could require additional knowledge. However, library code is usually much bigger than the application code, and iterating over it would lead to a huge overhead. In this context, `SARL` does not provide complex conditions, but it simply allows one to check the signature of a program member and annotate it. Therefore, ⟨LIB⟩ consists of the type ⟨T⟩ of program member we want to annotate, one or more annotations ⟨ANN⟩, together with the signature ⟨SIG⟩ of the target program member. When applied, this leads to adding the given annotations to all the program members whose signature fulfills the specified signature (definition *lib* in Figure 5, where *checkSignature* checks if two signatures represent the same element, and *typeOf* returns the type - class, field, method, or parameter - of a program component). Notice that, even if this component was specifically designed to operate on library code, it can be used also to annotate application code, avoiding the iteration on all program members by loading them through reflective calls.

Example Line 10 of WindowsForms specification (Figure 3) specifies to annotate with *@ResourceThatDoesNotNeedToBeClosed* all the getter methods of class *Brushes* that return a *Brush* instance, since these are system-wide objects handled by the runtime, and therefore should not be manually closed by the program.

`SARL` Application Figure 6 reports the algorithm for applying a `SARL` specification to a program. In particular, given a specification ⟨SARL⟩, and a program

Framework	Application	Version	Stars	Rank	LOCs	Time with	Time without
WindowsForms	Shadowsocks [4]	4.1.6	49768	1	12788	2'23"	2'14"
WindowsForms	ShareX [5]	12.4.1	13045	11	99191	5'24"	5'24"
WindowsForms	CefSharp [3]	73.1.130	7109	36	17863	2'01"	1'59"
ASP.NET	SignalR [2]	2.4.1	8067	18	49182	4'24"	4'36"
ASP.NET	AspnetBoilerplate [1]	4.5.0	8476	23	87288	6'15"	6'08"
ASP.NET	Umbraco [6]	8.0.2	2995	139	130384	11'09"	11'03"

Table 1. Analyzed applications

composed of an application a and a library l (both represented as set of classes), it applies the specification if and only if the rules set ⟨RULES⟩ is satisfied on the application a (line 4). If this is the case, it then sequentially applies all the specifications ⟨SPEC⟩ (lines 5-6), libraries ⟨LIB⟩ (lines 7-9), and implications ⟨IMPL⟩ (lines 10-12) contained in the `SARL` specification ⟨SARL⟩. Note that, while specifications ⟨SPEC⟩ are applied only to the application, library specifications ⟨LIB⟩ and implications ⟨IMPL⟩ are applied to both the application and the library part of the program. In this way, `SARL` allows adding information about the libraries of the framework, and not only to model the effects of the framework runtime model on the application.

5 Experimental Results

`SARL` has been interfaced with the Julia static analyzer, version 2.7.0.3, as specified in Figure 2. The `SARL` specification parser relies on JavaCC[5], while the semantics has been natively implemented in Java and passed to Julia through external (i.e., specified in an XML file rather than the application code) annotations.

In this Section, we analyze, for both WindowsForms and ASP.NET, the 3 most popular applications publicly available in GitHub that rely on these frameworks. We adopt as a metric of the popularity of a repository its number of stars. For each application we took the last stable release in the repository. All the statistics refer to the status of GitHub on June 28th, 2020. Each application has been analyzed with and without the framework specification. We report as lines of code (LOC) the number of physical lines of code reported by Locmetrics on the C# source files (with *cs* file extension) of the different applications in GitHub. Therefore, we consider only the code of the application, and not the libraries.

For each application, we compared the results of the analyses with and without `SARL`, investigating the number of warnings added or removed by the latter analysis. Each of such warnings has been manually investigated to ensure that no true positives were lost with our approach, and that new warnings can be accounted on the introduction of new entry points causing the analysis of previously unreachable code.

Table 1 reports the applications we selected, where column **Framework** reports the framework it uses, **Application** the name of the analyzed application, **Version** the analyzed version (taken from GitHub, thus directly associated with

[5] https://javacc.org/

	Shadow.	**ShareX**	**CefSharp**
Warn. w/o spec.	730	5471	465
Common (%)	473 (64.8%)	1397 (25.5%)	241 (51.8%)
Added (%)	0 (0%)	6 (0.1%)	0 (0%)
Removed (%)	257 (35.2%)	4074 (74.5%)	224 (48.2%)

Table 2. Difference in warnings on WindowsForms analyses

Warning	**SS**		**SX**		**CS**	
	A	**R**	**A**	**R**	**A**	**R**
ResourceNotClosedAtEndOfMethod	0	8	0	98	0	32
CloseableNotStoredIntoLocal	0	202	0	2802	0	124
FieldShouldBeReplacedByLocals	0	32	0	958	0	50
FieldIsOnlyUsedInConstructors	0	0	0	1	0	0
UselessAssignmentToDefaultValue	0	7	0	83	0	6
TestIsPredetermined	0	4	0	66	0	6
UnreachableInstruction	0	4	0	66	0	6
SetStaticInNonStaticWarning	0	0	6	0	0	0

Table 3. Warnings removed on WindowsForms applications

a commit that can be used for reproducibility), **Stars** the number of stars of the repository, **Rank** the rank of the repository among the C# ones, **Time with** and **Time without** the analysis time with and without the application of the framework specification (considering also the time needed for applying the specification to the application), respectively. All the analyses were executed on a r5.xlarge Amazon Web Service machine. These instances feature a Xeon Platinum 8000 series (Skylake-SP) processor with a sustained all core Turbo CPU clock speed of up to 3.1 GHz and 32 GB of RAM.

5.1 WindowsForms

Table 2 reports, for each application, the number of warnings with Basic checkers without the SARL specification, and the number of common, added and removed warnings when applying the specifications presented in Figure 3 of Section 4. The result highlight that even small specifications can have a major impact on the results of the analyses, removing a huge portion of false alarms issued due to the lack of knowledge by the analyzer. We will focus on the removed warnings only, since the 6 ones added in ShareX analysis are all real alarms which reside in methods that were previously considered dead code (thus not analyzed).

Table 3 reports the warnings removed (columns **R**) and added (**A**) when applying the WindowsForms specification to Shadowsocks (column **SS**), ShareX (**SX**) and CefSharp (**CS**), grouped by warning type. The specification targeted mostly disposable graphical objects stored into fields. The majority of the warnings (4403 out of 4556, that is, 96%) refers to these components, and we focus the following discussion on them. Appendix B.1 reports several examples of these warnings.

	SignalR	**ANB**	**Umbraco**
Warn. w/o spec.	681	552	1729
Common (%)	658 (96.6%)	544 (98.6%)	1658 (95.9%)
Added (%)	1 (0.1%)	0 (0%)	0 (0%)
Removed (%)	23 (3.4%)	8 (1.4%)	71 (4.1%)

Table 4. Difference in warnings on ASP.NET analyses

Warning	**SR**		**AB**		**UM**	
	A	**R**	**A**	**R**	**A**	**R**
FieldNeverUsed	0	6	0	0	0	1
FieldReadWritten	0	0	2	0	0	0
Uncalled	0	17	0	10	0	70
PossibleInsecureCookieCreation	1	0	0	0	0	0

Table 5. Warnings removed on ASP.NET applications

5.2 ASP.NET

Table 4 reports, for each analyzed application, the number of warnings with a standard analysis (without the SARL specification) executing Basic checkers all together, and the number of common, added and removed warnings when performing the same analyses with the specifications presented in Figure 1 of Section 1.1 (**ANB** is a shortcut for AspnetBoilerplate). It is noticeable that the results on ASP.NET applications are less pervasive that the ones on desktop applications: this is due to the nature of those projects which are rather libraries (SignalR and AspnetBoilerplate) or content providers (Umbraco), and they contain very few Web pages based on ASP.NET.

Table 5 reports the warnings removed (**R**) and added (**A**) when applying the ASP.NET specification to SignalR (**SR**), AspnetBoilerplate (**AB**) and Umbraco (**UM**) grouped by warning type. As for WindowsForms results, we will not discuss the added warnings since they are true alarms on methods that were previously considered dead code. Appendix B.2 reports several examples of these warnings.

6 Conclusion

In this paper, we introduced SARL, a domain specific language which allows one to specify the execution model of a framework. Such information is used to instruct a static analyzer about the model of execution of a framework. This approach enables the support of new frameworks through a readable and documentable model without modifying the code of the analysis engine, since it is applied to the analyzed application producing annotations that agnostically instruct the analyzer about the runtime environment. Furthermore, we applied this approach to six real-world applications dealing with two different frameworks, studying how the number of false alarms was reduced thanks to their respective specifications. Experimental result show that, even with extremely concise SARL specifications, all false alarms previously issued due to the lack of knowledge on these frameworks by the Julia static analyzer were successfully removed. The percentage of false alarms removed by SARL specifications highly varies depending on how much code relies on the frameworks (that is, from a minimum of 1.6% on AspnetBoilerplate to a maximum of 74.5% of ShareX).

Currently, we are working on the application of `SARL` to other frameworks, and in particular to Java Lombok [18], .NET Xamarin [26] and Unity [24].

References

1. Asp.net boilerplate, `https://github.com/aspnetboilerplate/aspnetboilerplate`
2. Asp.net signalr, `https://github.com/SignalR/SignalR`
3. Cefsharp, `https://github.com/cefsharp/CefSharp`
4. Shadowsocks for windows, `https://github.com/shadowsocks/shadowsocks-windows`
5. Sharex, `https://github.com/ShareX/ShareX`
6. Umbraco cms, `https://github.com/umbraco/Umbraco-CMS`
7. Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: Proceedings of ECOOP'13. Lecture Notes in Computer Science, Springer (2013)
8. Arzt, S., Bodden, E.: Stubdroid: automatic inference of precise data-flow summaries for the android framework. In: Proceedings of ICSE '16. IEEE (2016)
9. ASP.NET: (2018), `https://www.asp.net/`
10. Ball, T., Rajamani, S.: Slic: A specification language for interface checking (of c). Tech. Rep. MSR-TR-2001-21 (January 2002)
11. Centonze, P., Naumovich, G., Fink, S.J., Pistoia, M.: Role-based access control consistency validation. In: ISSTA (2006)
12. Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean Formulas for the Static Identification of Injection Attacks in Java. In: Proceedings of LPAR '15. Lecture Notes in Computer Science, Springer (2015)
13. Ferrara, P., Cortesi, A., Spoto, F.: Cil to java-bytecode translation for static analysis leveraging. In: Proceedings of FormaliSE '18. Springer (2018)
14. Forms, W.: (2018), `https://docs.microsoft.com/it-it/dotnet/framework/winforms/`
15. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39**(12) (2004)
16. J. Toman, D.G.: Concerto: a framework for combined concrete and abstract interpretation. In: Proceedings of the ACM on Programming Languages. vol. 3 (2019)
17. Leavens, G.T., Baker, A.L., Ruby, C.: Jml: a java modeling language. In: In Formal Underpinnings of Java Workshop '98 (1998)
18. Lombok: (2018), `https://projectlombok.org/`
19. Palsberg, J., Schwartzbach, M.I.: Object-Oriented Type Inference. In: Proceedings of OOPSLA '91. ACM Press (1991)
20. Spoto, F.: Nullness Analysis in Boolean Form. In: Proceedings of SEFM '08. IEEE (2008)
21. Spoto, F., Mesnard, F., Payet, E.: A Termination Analyzer for Java Bytecode Based on Path-Length. ACM Transactions on Programming Languages and Systems (TOPLAS) **32**(3), 1–70 (2010)
22. Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., Berg, R.: F4f: Taint analysis of framework-based web applications. In: Proceedings of the 2011 ACM International conference on Object-Oriented Programming, Systems, Languages, Languages, and Applications. vol. 16, pp. 1053–1068 (2011)
23. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web application. In: PLDI. ACM (2009)
24. Unity: (2018), `https://unity3d.com/`
25. Wikipedia: Software framework, `https://en.wikipedia.org/wiki/Software_framework`
26. Xamarin: (2018), `https://visualstudio.microsoft.com/xamarin/`

```
class MainForm : Form {
  ToolStripMenuItem tsmiTrayRecentItems;
}
class RecentTaskManager {
  void UpdateTrayMenu() {
    ToolStripMenuItem tsmi = MainForm.tsmiTrayRecentItems;
    ToolStripMenuItem tsmiLink = new ToolStripMenuItem();
    if  (...)  tsmi.DropDownItems.Insert(2, tsmiLink);
    else tsmi.DropDownItems.Add(tsmiLink);
  }
}
```

Fig. 7. Disposable objects stored in fields of IComponent classes

A Additional Tables

Member	**Trg**	**Operators**
Class	*cls*	**name**, subtypeOf, containsMethod, containsField, hasAnnotation
Field	*fld*	definingClass, **name**, type, basicType, hasAnnotation, hasAccessor
Method	*mtd*	definingClass, **name**, returnType, basicReturnType, hasLocalVariable, hasAnnotation, numberOfParameters, hasAccessor, hasParameter
Method Parameter	*par*	definingMethod, **name**, index, type, basicType, hasAnnotation
Local Variable	*var*	definingMethod, **name**, type, basicType
Annotation	*ann*	**name**, hasOptionValue
String	*str*	**equals**, contains, startsWith, endsWith, matches
Integer	*int*	**equals**

Table 6. Targets for conditions, with their respective operators (default operators are in bold)

B Examples from Experimental Results

B.1 WindowsForms

CloseableNotStoredIntoLocal and ResourceNotClosedAtEndOfMethod Warnings about closable resources are issued each time an object that implements *Closeable* (in Java) or *IDisposable* (C#) might not get closed/disposed: such an object should be stored in (i) a *final/readonly* field on which a call to *close()/Dispose()* happens in reachable code, or (ii) a local variable on which a call to *close()/Dispose()* happens before the end of the method.

Figure 7 shows a snippet of code from ShareX. In *RecentTaskManager. UpdateTrayMenu()* a new *ToolStripMenuItem* is created and then added to another *ToolStripMenuItem* retrieved from a field of *MainForm*. Since (i) *ToolStripMenuItem* implements *IComponent*, (ii) *MainForm* inherits from *Form* which implements *IComponent*, and (iii) the newly created *ToolStripMenuItem* will be

Operator	**Trg**	**What does it check**
Non-terminal operators		
definingMethod	*mtd*	current member's containing method
name	*str*	current member's name
index	*int*	current parameter's index
type	*cls*	type of the current member
basicType	*str*	type of the current member
hasAnnotation	*ann*	current member's annotations
definingClass	*cls*	current member's containing class
returnType	*cls*	return type of the current method
basicReturnType	*str*	return type of the current method
hasVariable	*var*	current method's local variables
hasAccessor	*str*	current member's accessors
numberOfParameters	*int*	current method's number of parameters
hasParameter	*par*	current method's parameters
subtypeOf	*cls*	current class' superclasses, interfaces (recursively)
containsMethod	*mtd*	current class' methods
containsField	*fld*	current class' fields
hasOptionValue	*str*	current annotation's specified member's value
Terminal operators		
equals		the current string/integer is equal to the given value
startsWith		the current string starts with the given value
endsWith		the current string ends with the given value
contains		the current string contains the given value
matches		the current string matches the given regular expression

Table 7. Operators' details

reachable from *MainForm* after the execution of the if-else block, such object will be automatically disposed from the runtime environment when the instance of *MainForm* will be disposed by the WindowsForms runtime. When analyzing ShareX without the WindowsForms specification, Julia raises the following warning:

RecentTaskManager.cs:156: [CloseResource] *This instance of class "ToolStripMenuItem" does not seem to be always closed by the end of this method. It seems leaked at line 156*

When we apply the WindowsForms specification in Figure 3 to this code, line 8 annotates field *MainForm.tsmiTrayRecentItems* as *@AutoClosedResource*, and this informs Julia that every resource reachable from that field will be automatically disposed. Hence, the above warning will not be issued anymore, since the newly created *ToolStripMenuItem tsmiLink* will end up being reachable from such field.

FieldShouldBeReplacedByLocals and FieldIsOnlyUsedInConstructors These kinds of warnings are issued when a field could be replaced by a local variable inside the *only* method (FieldShouldBeReplacedByLocals) or constructor (FieldIsOnlyUsedInConstructors) that references them.

```
class StatisticsStrategyConfigurationForm : Form {
  Button OKButton;
  void InitializeComponent() {
    OKButton = Button();
    // init code
    splitContainer1.Panel2.Controls.Add(OKButton);
  }
}
```

Fig. 8. UI fields generated by Visual Studio

```
class BrowserTabUserControl {
  IContainer components = null;
    override void Dispose(bool disposing) {
      if (disposing) {
        if (components != null) {
          components.Dispose();
          components = null;
        }
      }
    base.Dispose(disposing);
  }
}
```

Fig. 9. Dispose() pattern of Form classes

Figure 8 shows a simplification of code generated by the WindowsForms framework to represent a UI component in Shadowsocks. The field *OKButton* is initialized with a button instance, and added to a container inside *Initialize-Component()*, but the field is never used later in the code. Such initialization is located inside the Visual Studio designer-generated file, and the user has no responsibility for this. Julia raises the following warning on this code:

StatisticsStrategyConfigurationForm.cs: [ImproperField] *Field "OKButton" should be replaced by local variables*

When we apply the WindowsForms specification in Figure 3 to this snippet of code, line 7 annotates field *OKButton* as ExternallyRead and Injected, and Julia does not produce the warning above anymore.

TestIsPredetermined, UnreachableInstruction and UselessAssignment-ToDefaultValue Julia's analysis is able to detect when a test always evaluates to true or false. In this situation, two warnings are issued: the first stating that the test is useless since it always evaluates to the same Boolean value, and the second one to explicitly mark the unreachable branch (if this contains some code). In addition, when a field or a local variable gets initialized with its default value Julia raises an UselessAssignmentToDefaultValue warning.

Figure 9 shows a pattern generated by Visual Studio for handling the disposal of resources of *Form* classes in CefSharp. The *components* field is non-null only if the form contains some resources that are not UI objects but needs to be disposed (e.g., a *Timer* instance). However, the field is initialized and managed

```
abstract class AbpWebApplication : HttpApplication {
  void Application_Start (object sender, EventArgs evargs) {
    // startup code
  }
}
class MvcApplication : UmbracoApplicationBase { }
```

Fig. 10. Application_Start method

by the framework runtime, and the code of the application never assigns it. The code of the designer declares the field, initializes it to *null*, and then disposes it if it is not null. On this piece of code, Julia raises the following three warnings:

BrowserTabUserControl.cs:8: [UselessAssigment] *Useless assignment of field "components" to its default value*
BrowserTabUserControl.cs:103: [UselessTest] *The result of this test is fixed: you are comparing null against null*
BrowserTabUserControl.cs:105: [Deadcode] *This instruction seems unreachable*

When we apply the WindowsForms specification in Figure 3 to this snippet of code, line 9 annotates field *components* as externally injected (as it effectively happens in the framework runtime), and therefore Julia analyses do not produce anymore these warnings.

B.2 ASP.NET

Uncalled Julia issues a warning on each method that is not reachable from the entry points of the application, and the code from these methods is never analyzed.

Figure 10 shows an *Application_Start* method in AspnetBoilerplate. While this is never actually used within the application code, it is indeed invoked by the framework runtime at the first startup of *MvcApplication*. Hence, even if its access is restricted (i.e., it is not *public*), it must be considered as an entry point.

Besides, the compilation of the web views of the application results in an assembly containing one class file per view (named *ASP._Page_<namespace>_<viewName>*) with only a constructor, a getter for the current application instance and an *Execute* method, and an object factory (named *_ASP.FastObjectFactory_<applicationName>*) that is used by ASP.NET to instantiate the web views. These methods are both (i) generated code, and (ii) invoked by the runtime.

When analyzing Umbraco, Julia raises the below warnings:

UmbracoApplicationBase.cs:72: [Deadcode] *Method "Application_Start" is unreachable*
__ASP.FastObjectFactory_umbraco.cs: [Deadcode] *Method "Create_ASP__Page_Umbraco_Install_Views_Index_cshtml" is unreachable*
ASP._Page_Umbraco_Install_Views_Index_cshtml.cs: [Deadcode] *Method "get_ApplicationInstance" is unreachable*

```
class _Default : Page {
  TextBox userName;
  TextBox roles;
  Button login;
  void Login(object sender, EventArgs e) {
    var identity = new GenericIdentity(userName.Text);
    var principal = new GenericPrincipal( identity ,
          SplitString ( roles . Text ));
  }
}
```

Fig. 11. UI fields generated by Visual Studio

When we apply the ASP.NET specification in Figure 1 to this snippet of code, line 8 annotate *Application_Start* as an entry point, while lines 10 and 11 do the same for the latter methods, removing all three warnings from the results.
FieldNeverUsed This type of warning is issued whenever a field is never read or written inside the whole reachable application code. Figure 11 shows a snippet of ASP.NET code from application SignalR. Method *Login* is an event handler, hence is never explicitly called in the reachable code. In addition, fields *userName* and *roles* are never explicitly initialized: they represent an alias for the web view component declared in the *cshtml* file, and their values will be injected from the runtime environment. When analyzing SignalR without the SARL specification, Julia raises the following warnings:

```
_Default.cs: [FieldAccess] Field "userName" is never used
_Default.cs: [FieldAccess] Field "roles" is never used
_Default.cs: [FieldAccess] Field "login" is never used
_Default.cs: [Deadcode] Method "Login" is unreachable
```

When we apply the ASP.NET specification in Figure 1 to this code, line 8 annotates method *Login* as an entry point, while line 9 annotates fields *userName*, *roles*, and *login* as externally read and injected, thus removing the four warning reported above.