Formal comparison of CrySL with jGuard

Anonymous

March 31, 2022

1 Runtime semantics of actions:

First, we introduce a formal specification for how actions impact instance guards. This is also required to follow the formal constructions used. An analogous construction can be made for static guards [1].

Let G_i be the set of all declared instance guards in a program. We model the state of instance guards as a function I mapping Java objects to a subset of G_i. Here, we say that a guard $g_i \in G_i$ is set for an object o iff $g_i \in I(o)$. With this notion, an instance guard g on an object o evaluates to `true` iff $g \in I(o)$ for a given guard state I. Executing an **Action** is defined as a state transition $f_g : I \rightarrow I'$. If g is an instance guard reference, let o be the Java object on which the method was called. Then, we define I' as:

$$I' : (O \rightarrow \mathcal{P}(G_i) : x \mapsto \begin{cases} I(x) \cup \{g\} & \text{if } x = o \text{ and is set} \\ I(x) \setminus \{g\} & \text{if } x = o \text{ and is reset} \\ I(x) & \text{otherwise} \end{cases}$$

This construct allows us to specify the state transition for an instance guard, $f_g(I) = I'$. When a new object o of a verified class is instantiated, a similar state transition defines $I(o)$ as the set of all `initially set` guards to represent the object's initial guard state.

Essentially, every action modifies the state of instance guards to indicate which guards are currently set for an object.

2 CrySL

CrySL can introduce typed objects, which are essentially variables that are bound through method invocations. Events refer to methods in the specified class, using previously defined objects to describe arguments or return values. In

[1] We ignore static guards for this part, because CrySL does not support a comparable mechanism.

```
class Counter {
  public Counter() {}
  public Counter(int startValue) {}

  public void increment(int by) {}
  public void decrement(int by) {}
  public void take(Counter counter) {}
  public int freeze();
}
```

```
SPEC  org.example.Counter

OBJECTS
  int init;
  int incBy;
  int decBy;
  org.example.Counter taken;
EVENTS
  i1:   org.example.Counter();
  i2:   org.example.Counter(init);
  inc:  increment(incBy);
  dec:  decrement(decBy);
  take: take(taken);
  done: freeze();
  init: i1 | i2;
  change: inc | dec | take
ORDER init, change*, done;
CONSTRAINTS
  inc > 0;
  dec > 0;
  notNull[taken];
REQUIRES
  finalized[taken];
ENSURES
  finalized;
```

Figure 1: Exemplary Java class and a matching CrySL rule

a `CONSTRAINT` section, simple expressions may then be used to describe stateless constraints about variables declared under `OBJECTS`. The `ORDER` clause defines a regular expression of events describing allowed sequences of invoked methods. Finally, the `REQUIRES` and `ENSURES` mechanism declares stateful pre-conditions referring to other objects.

As an example, consider the Java class and a matching specification in Figure 1. It describes a simple counter which can be incremented or decremented. After changing its value through those calls, a `freeze` method returns the value of the counter and forbids further operations changing the counter. A `take` counter sets the value of the counter to the value of another counter on which `freeze` must have already been called. A counter can be initialized with or without a value.

The corresponding CrySL rule starts by declaring variables for all parameters used in events described later. Each method from the class is described with an event. Further, the two constructors and the three methods changing the counter are grouped into an aggregate event (`change`) for simplicity. Next, an `ORDER` section describes valid usage sequences in regex format, which start by initializing a counter, changing its values and finally calling `freeze`. A requires and ensures section describes that the parameter to `take` must be a finalized counter. We use an example where the same CrySL rule `REQUIRES` and `ENSURES` the same predicate, but in practice predicates that are required by one rule are ensured by another, thereby enabling expression of cross-object rules. Finally, a `CONSTRAINT` section ensures that the parameters to `increment` and `decrement` are positive integers.

A full definition of CrySL and its formal semantics is given by Krüger et al. [?]. Here, we describe map misuse categorization to CrySL concepts. To the best of our knowledge, currently no other DSL is able to express as many misuse categories as CrySL.

- OBJECTS and EVENTS gather parameters and methods that can be violated for a particular type. These allow expressing **Unordered usage patterns**.
- ORDER describes in what order methods of a particular type must be invoked. This allows one to express **sequential usage patterns**.
- CONSTRAINTS describe the allowed range of values for parameters to method calls. This allows one to express **Behavioral specifications/ Conditions**.
- ENSURES AND REQUIRES clauses describe how objects of different types must be correctly composed. By rely/guarantee-reasoning, they allow one to express the **multi-object properties** which span multiple API objects.

Now, we show that for every CrySL rule, we can construct equivalent jGuard annotated code building upon runtime semantics of guards described in Section . In this paper, we will introduce the construction for The `ORDER`. We have omitted the proofs for the other sections for lack of space. If accepted, we will make available publicly the proof for similar constructions for the `CONSTRAINTS` and `ENSURES/REQUIRES` predicates.

2.1 Expressing CrySL's Order in JGuard

The `ORDER` section of a CrySL rule corresponds to a DFA $\mathcal{A}^o = (Q, \mathbb{M}, \delta, q_0, F)$ with $Q = \{q_0, \ldots, q_n\}$ defining a language $L(\mathcal{A}^o) \subseteq \mathbb{M}^*$ of allowed method sequences [?]. We construct a verified implementation of a class for every such DFA and complete the construction by establishing that every misuse reported by the DFA will also be reported by the constructed verified class (by showing semantic equivalence between the constructed verified class and the corresponding CrySL DFA).

Construction of a verified class corresponding to a CrySL DFA: Each state in the CrySL DFA can be represented by a private guard added to the class. Depending on the state q_i, the guard is declared as follows:

1. Each state q_i declares a private guard named g_i
2. If $i = 0$, the guard g_0 is declared to be `initially set`
3. If $q_i \notin F$, the guard g_i is declared to be `finally reset`

Next, we define changes to each verified method $m \in \mathbb{M}$ necessary to implement state transitions defined in the DFA of the CrySL rule $\mathcal{A}$. The method m is replaced with a verified method, copying its original visibility, return type, type parameters, parameters, thrown types and body. For each pair $(q_i, q_j) \in Q^2$ with $\delta(q_i, m) = q_j$, a **Consequence** is added to m. The consequence has a **ConditionTrigger** matching q_i and changing the currently active guard from g_i to g_j. In jGuard, the syntax for such consequence is `when` g_i `resets` g_i `,sets` g_j.

Semantic equivalence of the constructed verified class and the CrySL DFA: To show that the semantics of the constructed verified class matches the runtime semantics of the CrySL DFA, let m^o be a sequence of method invocations on an instance of the annotated class. Further, let $(s_0, \ldots, s_{n-1}) \in Q^+$ be the trace of states taken in $\mathcal{A}$ when running over m^o. That is, $s_0 = q_0$ and $\delta(s_i, m_i) = s_{i+1}$ for all $0 \leq i < |m^o| = n$. Per induction, we show that for each such i, $s_i = q_j$ iff the guard g_j is set. As mentioned in the runtime semantics of declaring guards, for $i = 0$, the initial state is q_i and the only guard set is g_i. For $i > 0$, the automata is in the state $s_i = q_j$ if it has been in the state $s_{i-1} = q_k$ before the method invocation m_i. As per the induction hypothesis, this implies that g_k was the only guard active before the method invocation m_i. When the method returns, its consequences are evaluated. As g_k is the only guard set, only the consequence introduced for $\delta(q_k, m_i) = q_j$ will match. This consequence then resets g_k and sets g_j, satisfying the induction hypothesis. Having established that the guards in a method sequence match states taken by $\mathcal{A}$, we must show that a misuse is reported iff $s_{n-1} \notin F$ (the final states). If $s_{n-1} = q_f$ is in F, the guard g_f is the only guard active at the end of the method invocation trace. As g_f has been constructed without a `finally` clause, no further checks are performed by jGuard and no misuse is reported. On the contrary, if $s_{n-1} = q_e \notin F$, the single set guard g_e has been declared to be `finally reset`. The semantics of jGuard mandate a misuse being reported in this case.

As the construct shown here reports a misuse iff $m^o \notin L(\mathcal{A})$ (method sequence is not valid in the language defined by the CrySL's DFA), it matches the runtime semantics of CrySL.

2.2 Expressing CrySL's Constraints in JGuard

Here, we will define a mapping from every CrySL constraint to a corresponding jGuard expression which is then used in an appropriate requirement.

In CrySL, a constraint is a boolean expression on specified objects. Specified objects are declared as variables. Variables are then bound in response to an event. Each set of bound variables must adhere to all constraints specified in a rule. One does not need to evaluate all constraints for each event to adhere to CrySL's formal semantics. An equivalent check is to evaluate only those constraints referencing variables that might have changed. We now construct a mapping τ mapping CrySL constraint c to jGuard expressions. Using this expression in a **Requires** section for the method m will then yield an equivalent check. For each method m mentioned as an event in a CrySL rule, let C_m be the set of constraints. For each $c \in C_m$, we now define $\tau(c)$ as follows[2].

- If c is of the form `c`$_1$ `=>` `c`$_2$, then $\tau(c) = !\tau(c_2)|\tau(c_1)$. This is equivalent, as $a \implies b \iff \neg a \vee b$.
- If c is of the form `c`$_1$`<OP>c`$_2$ with ¡OP¿ being `||`, `&&`, `+`, `-`, `%`, `*`, `/`, `<`, `<=`, `>`, `>=`, `!=` or `==` then $\tau(c) = \tau(c_1)\langle OP\rangle\tau(c_2)$.
- If c is of the form `c`$_1$`.ID` for an identifier ID, then $\tau(c) = \tau(c_1).ID$
- If c is of the form `elements(c`$_1$`) in (c`$_2$`, ..., c`$_k$`)`, then $\tau(c) =$ `Arrays.asStream(c`$_1$`).allOf(el ->` $\tau($ `el in (c`$_2$`, ..., c`$_k$`)` $))$
- If c is of the form `!c`$_1$, then $\tau(c) = !\tau(c_1)$
- If c is of the form `(c`$_1$`)`, then $\tau(c) = (\tau(c_1))$
- If c is a string, integer or boolean literal, $\tau(c) = c$
- If c is of the form `c`$_1$ `in (c`$_2$`, ..., c`$_k$`)`, then $\tau(c) =$ `Arrays.asList(`$\tau(c_2), \dots, \tau(c_k)$`).contains(`$\tau(c_1)$`)`.
- If c is a variable reference and the method m has a parameter with the same name p, then $\tau(c) = p$.
- If c is of the form `instanceof[c`$_1$`, CLS]`, then $\tau(c) = \tau(c_1)$ `instanceof CLS`. Similarly, `neverTypeOf[obj, CLS]` is translated to the logical inverse.
- If c is of the form `length[c`$_1$`]`, $\tau(c) = \tau(c_1)$ `.length`
- c is of the form `callTo[ID]`, consider an event e with the name `ID`. Next, obtain a state machine $\mathcal{A}'$ based on the DFA defined in section 2.1 with seperate states depending on whether the e has occurred or not. That is, all states are duplicated with equal state transitions between them, but all state transitions for e lead to $\mathcal{A}'$ being in a different state set. Now, c can be transformed into the Java expression $g_1|| \dots ||g_n$, where g is the sequence of guards introduced for the states active after a call to id. Similarly, `noCallTo[ID]` can be compiled by checking for guards introduced for the original set of states.
- CrySL allows extracting parts of a cipher transformation (*algorithm/mode/padding*) passed to `Cipher.getInstance` these parts can be extracted by matching the inner value against a regular expression, which can be done as a single Java expression.

[2]We have listed a construction for a few CrySL constraints and omitted the rest for lack of space

Next, each method m with $C_m \neq \emptyset$ is turned into a verified method. All requirements $\tau(c) \mid c \in C_m$ are added to the verified method. This yields a construction semantically equivalent to the CrySL rule.

CrySL supports a predicate `notHardCoded(`e`)`, which evaluates to `true` iff the expression e is not hard coded in the program's source, e.g. through a string literal. As this cannot be detected at runtime reliably, jGuard does not have a comparable mechanism and this predicate cannot be translated. However, it should be noted that the `notHardCoded` predicate in CrySL cannot be fully accurate either. It is designed to express guards against static cryptographic keys embedded in the application. Still, it would fail to detect constants being loaded dynamically, for instance by reading a file contained in the application's JAR through a class loader. These values are still conceptually hard coded, without being detectable by a CrySL rule.

2.3 Expressing CrySL's Ensures and Requires in JGuard

As defined in Section 2, the `ENSURES` section sets or resets a predicate, identified by the class annotated by a CrySL rule and a name for the predicate. Additionally, a predicate may hold arguments, which will be bound to variables in CrySL. Each predicate is defined on an instance o of the class C annotated by the CrySL rule. Further, each predicate has a name and a set of additional arguments $(a_0, \ldots, a_i)$.

To express CrySL predicates in jGuard, a new class and an external state declaration is introduced. The class contains a field for each parameter a_i in the predicate. The external state declaration is defined on this class. As an example, consider the generated class for the `finalized` predicate from Figure 1:

```
class CounterFinalized {}

state CounterFinalized for Counter {
  Set<CounterFinalized> finalized = Collections.emptySet();
}
```

By default, predicates declared in an `ENSURES` section are added or removed in the last event of valid method sequences. In addition, CrySL allows specifying an `after` clause to perform the change after an event. As all events can be mapped to a state q in the automaton $\mathcal{A}$ as defined in Section 2.1, we must add or remove valid predicates in response to $\mathcal{A}$ entering such state.

To set a predicate with argument variables $v_0, \ldots, v_i$ in state q, consider all tuples $(q_p, m, q) \in Q \times \mathbb{M} \times Q$ with $\delta(q_p, m) = q$. As constructed in Section 2.1, the equivalent verified method will have a **Consequence** set for this state transition. To represent a predicate being fulfilled, we add an action that sets the state variable to this consequence. This consequence shall construct a new instance of the predicate class with the bound variables. Note that all CrySL variables available in this construct refer to method parameters or its return value, which means that they can be transformed to jGuard expressions available in a consequence. The state variable is then changed to a new set containing previous predicates and the new instance.

To remove an active predicate, the corresponding consequence simply sets the introduced state variable to a new set not including the predicate to be removed.

With ensures clauses being translated to external state declarations that are set iff the corresponding CrySL predicate is set, a `REQUIRES` section can be translated to a jGuard requirement. The requirement simply queries the external state declaration on the object to verify whether a matching predicate exists or not. An absence of this predicate will cause the requirement to evaluate to `false`, causing a misuse to be reported.