Specifying Robustness

AUTHOR1, UnivOne
AUTHOR2, UnivTwo
AUTHOR3, At the awesome Centre
AUTHOR1, UnivOne
AUTHOR2, UnivTwo
AUTHOR3, At the awesome Centre

We argue that the specification of what makes a program robust goes beyond ...[1]

Only somebody with the bank of a currency can violate conservation of that currency

In this short note we revisit the specification of such properties. We introduce new fundamental OCAP assertions $\mathcal{A}ccess$(,a)nd , [2] and use them to specify protection policies.

I believe that the new fundamental OCAP assertions overcome the problems we had identified with the old specs we had written for protection policies.

ACM Reference Format:
author1, author2, author3, author1, author2, and author3. 2018. Specifying Robustness. 1, 1 (November 2018), **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 THE PROPOSAL

I propose here that we use 1a) and 2b). The novelty is that in the formulation of "can violate conservation of that currency", we consider not only who caused the violation (*i.e.* who was the receiver of the method execution which eventually modified the currency), but we also consider the *set of objects* which were *involved* in the execution of that method (*i.e.* the objects whose fields were read or written, or which executed methods, or whose identity was used in some way during that execution). This allows us to differentiate between the object which caused the change in the currency, and the object which had the *direct* access to the bank.

To express the protection policy for the currency we will mandate that at least *one* of the objects involved had direct access to the bank, that this access existed already at the start of that method call, and that the object which had access to the bank object is not a `account` object.

[1]Is the title too ambitious?
[2](to replace the old definitions of $\mathcal{M}ay\mathcal{A}ccess$ and)

Authors' addresses: author1UnivOne; author2UnivTwo; author3At the awesome Centre; author1UnivOne; author2UnivTwo; author3At the awesome Centre.

XXXX-XXXX/2018/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
class Bank {
  private field ledger; // a Node

  Bank( ){  ledger = null; }

  fun makeAccount(amt){  ledger = new Node(Account(this), amt, ledger); }
}

class Account {
  private field myBank;

  Account(aBank){  myBank = aBank;  }

  fun sprout( )
  // create Account in same Bank with 0 balance

  fun deposit(source,  amnt)
  // if source and receiver are in same Bank,  and source holds enough money,
  //  then transfer amnt from source into receiver
  { ...  }
}

 class Node{
   field myBalance; // the balance, a number
   field myAccount // the account
   field next; // the next node

   ...
}
```

Fig. 1. The Bank example – outline

2 THE BANK

Remember the system for electronic money proposed in [?]. In Fig. **??** we outline one possible implementation, where the accounts keep a reference to their bank, the bank keeps a list of pairs of accounts and balances. We use a class `Node` in order to make a particular point later. Fields declared `private` can only be read or wrriten by the object itself. Note that the fields of class `Node` are not private; this is so because `Node` objects do not "escape" the module `Bank`/`Account` and need not be as robust as `Bank` objects or `Account` objects.

Figure **??** contains a diagram of some objects from the classes `Bank`, `Account` and `Node`, as well as some external objects of unknown provenance (in grey).

3 THE PROTECTION POLICIES

Remember the five policies proposed in [?]. In this note, we only look at two of them:

Pol_2 Only someone with the bank of a given currency can violate conservation of that currency.
Pol_4 No one can affect the balance of a account they don't have.

Fig. 2. Diagrammatic representation of some objects from `Bank`, `Account` *etc.*

4 PERMISSION AND AUTHORITY

Policy **Pol_2** ties *authority* with *permission*: namely, permission to access the bank is a necessary condition for authority over the bank's currency. We will define the OCAP assertions $\mathcal{A}ccess(,()$permission) and (authority). [3] Their intuitive meaning is as follows:

- $\mathcal{A}ccess((,)\, y, \texttt{z})$ means that in the current runtime configuration, the object indicated by $\texttt{y}$ has *direct* access to the object indicated by $\texttt{z}$. This direct access is given either because $\texttt{z}$ is one of $\texttt{y}$'s fields, or because $\texttt{z}$ is one of the arguments or local variables in the method body currently executing and that $\texttt{y}$ is the receiver.
- $(\texttt{x}, \texttt{e}, \texttt{S})$ means the current receiver is $\texttt{x}$, and that execution of the current configuration will eventually change the value of $\texttt{e}$, and that this execution will only involve (ie call methods on, read or write fields from) objects from the set $\texttt{S}$.

For example, in the runtime configuration from Figure **??**, and assuming that $\texttt{x1}$, $\texttt{x10}$, $\texttt{x11}$ are local variables mapping to addresses $\texttt{1}$, $\texttt{10}$, and $\texttt{11}$, then we have that $\mathcal{A}ccess((,)\, x11, \texttt{x10})$, and $\mathcal{A}ccess((,)\, x10, \texttt{x1})$, but $\neg\mathcal{A}ccess((,)\, x11, \texttt{x1})$.

The assertion only holds if the set S includes all objects involved in causing the change of the value of $\texttt{e}$. For example, still in Figure **??**, if the classes of the objects at $\texttt{x10}$ and $\texttt{x11}$ contain appropriate methods, and if the current receiver is $\texttt{11}$, then it is possible that $(\texttt{x11}, \texttt{x1}.Currency, \{\texttt{x11}, \texttt{x10}, \texttt{x1}\},)$, but regardless of the code in these objects, we have $\neg(\texttt{x11}, \texttt{x1}.Currency, \{\texttt{x11}, \texttt{x10}, \texttt{x1}\},)$

Definitions and Naming Conventions We now proceed with the precise definitions. Remember first the naming conventions that $\texttt{M}$ stands for a module (ie class definitions), that σ stands for a runtime configuration (ie currently executing sequence of statements with frames and heap), that $\texttt{e}$ stands for an expression, that $\lfloor \texttt{e} \rfloor_{\texttt{M},\sigma}$ is the value of the expression $\texttt{e}$ in the state σ, and that $\texttt{M} \vdash \sigma \leadsto \sigma'$ expresses that execution of runtime configuration σ in the context of the class definitions from module $\texttt{M}$ leads in one small step to σ'.

DEFINITION 1 (PERMISSION AND AUTHORITY). *Given a module* $\texttt{M}$*, identifiers* $\texttt{x}$ *and* $\texttt{y}$*, expression* $\texttt{e}$*, and runtime configuration* σ*, and a set of addresses* S*, we define validity of the assertions* $\mathcal{A}ccess(,a)nd$ *as follows:*

- $\texttt{M}, \sigma \models \mathcal{A}ccess((,)\, x, \texttt{y})$ *iff*
  - $\sigma(\texttt{x}, \texttt{f}) = \sigma(\texttt{y})$ *for some field* $\texttt{f}$*, or*
  - $\sigma(\texttt{this}) = \sigma(\texttt{x})$ *and* $\sigma(\texttt{z}) = \sigma(\texttt{y})$*, for some some parameter of local variable* $\texttt{z}$.
- $\sigma\ |_S$ *denotes a* restriction *of* σ *to the objects from the set* S*. That is, the domain of the heap in* $\sigma\ |_S$ *is* S*, and otherwise,* $\sigma\ |_S$ *is identical to* σ.
- $\texttt{M}, \sigma \models (\texttt{x}, S, \texttt{e})$ *iff*
  - $\sigma(\texttt{this}) = \sigma(\texttt{x})$ *and*
  - $\exists \sigma'.\ \texttt{M} \vdash \sigma\ |_S \leadsto^* \sigma'$*, and* $\lfloor \texttt{e} \rfloor_{\texttt{M},\sigma} \neq \lfloor \texttt{e} \rfloor_{\texttt{M},\sigma'}$.
- $\texttt{M}, \sigma \models \mathcal{W}ill\mathcal{A}ccess\mathcal{T}hrough(\texttt{x}, S, \texttt{y})$ *iff*
  - $\exists \sigma'.\ \texttt{M} \vdash \sigma\ |_S \leadsto^* \sigma'$*, and* $\texttt{M}, \sigma' \vdash \mathcal{A}ccess((,)\, x, \texttt{y})$.

We can easily prove the following lemma:

LEMMA 4.1. *For sets* $\texttt{S}$ *and* $\texttt{S}'$*, runtime configuration* σ*, variable* $\texttt{x}$ *and expression* $\texttt{e}$*, if* $\sigma \models \texttt{S} \subseteq \texttt{S}'$*, then*

- $\sigma \models (\texttt{x}, \texttt{e}, \texttt{S})$ *implies* $\sigma \models (\texttt{x}, \texttt{e}, \texttt{S}')$.
- $\sigma \models \mathcal{W}ill\mathcal{A}ccess\mathcal{T}hrough(\texttt{x}, \texttt{y}, \texttt{S})$ *implies* $\sigma \models \mathcal{W}ill\mathcal{A}ccess\mathcal{T}hrough(\texttt{x}, \texttt{y}, \texttt{S}')$.

[3] Note that they are slightly different assertions to those we had in the past.

5 INVARIANTS

We define below the meaning of invariants.[4] The assertion $\texttt{M} \models A$ requires that the assertion A is satisfied in all reachable states. The set $\mathcal{A}rising(\texttt{M})$ contains all runtime configurations which can be reached when starting with an empty heap, and executing any expression consisting of constructor and method calls as defined in M.[5] The term $\texttt{M} * \texttt{M}'$ denotes the result of linking two modules – the operation is defined only when the two modules do not have overlapping definitions.

DEFINITION 2 (INVARIANTS). *For a module* M *and assertion A we define:*

- $\texttt{M} \models A$ *iff* $\forall \texttt{M}'.\, \forall \sigma \in \mathcal{A}rising(\texttt{M}' * \texttt{M}).\ \texttt{M}' * \texttt{M}, \sigma \models A$

The use of the set of configurations from $\mathcal{A}rising(\texttt{M}' * \texttt{M})$ reflects that policies need to hold in an *open* world, where we link against *any* module $\texttt{M}'$, about which we know nothing.

6 SPECIFYING POL_2 AND POL_4

We give a formal definition of **Pol_2** and **Pol_4**, using the concepts defined earlier in Definition **??**:

DEFINITION 3. *We define* **Pol_2** *and* **Pol_4** *as follows:*

$$\textbf{Pol\_2} \equiv \forall \texttt{b}.\forall \texttt{o}.\forall \texttt{S}.\,[\ \ \texttt{b}:\texttt{Bank} \wedge \texttt{b} \neq \texttt{o} \wedge (\texttt{o}, \texttt{b.Currency}, \texttt{S}) \longrightarrow \exists \texttt{o}'.\,[\,\texttt{o}' \in \texttt{S} \wedge \mathcal{A}ccess((,)\,o', \texttt{b}) \wedge \neg(\texttt{o}':\texttt{Account})\,]\ \ ]$$

$$\textbf{Pol\_4} \equiv \forall \texttt{a}.\forall \texttt{o}.\forall \texttt{S}.\,[\ \ \texttt{a}:\texttt{Account} \wedge \texttt{a} \neq \texttt{o} \wedge (\texttt{o}, \texttt{a.Balance}, \texttt{S}) \longrightarrow \exists \texttt{o}'.\,[\,\texttt{o}' \in \texttt{S} \wedge \mathcal{A}ccess((,)\,o', \texttt{a}) \wedge \neg(\texttt{o}':\texttt{Account} \cup \texttt{Bank})\,]\ \ ]$$

Pol_2 guarantees that if an object o≠b may affect the value of `b.Currency` only if the objects involved in the process of affecting the value of `b.Currency` include at least an object o′ which had direct access to b, and whose class is not `Account`. Stated positively, this policy mandates that exporting an `Account` to an environment will not affect the `Currency` of b. In other words, `Account`s protect the integrity of the `Bank`'s currency.

In more detail, by applying Definition **??** on Definition **??**, the meaning of policy **Pol_2** is, that a runtime configuration σ satisfies **Pol_2** if whenever the current receiver in σ is not a `Bank` object, and the execution of σ leads to another runtime configuration σ' with a different value for `b.Currency`, then the objects involved in the execution from σ to σ' include at least one object which had direct access to b. Note that this direct access needs to exist at the beginning of the execution, *i.e.* at σ. Formally:

$$\texttt{M}, \sigma \models \textbf{Pol\_2} \longleftrightarrow \forall \texttt{b}.\forall \texttt{o}.\forall \texttt{S}.\,[\ \ \texttt{M}, \sigma \models \texttt{b}:\texttt{Bank} \wedge \sigma(\texttt{b}) \neq \sigma(\texttt{o}) \wedge \sigma(\texttt{this}) = \sigma(\texttt{o}) \wedge \exists \sigma'.(\ \texttt{M} \vdash \sigma\,|_{\texttt{S}}\!\rightsquigarrow^{*} \sigma' \wedge \lfloor \texttt{b.Currency} \rfloor_{\texttt{M},\sigma} \neq \lfloor \texttt{b.Currency} \rfloor_{\texttt{M},\sigma'}\,) \longrightarrow \exists \texttt{o}'.(\ \sigma(\texttt{o}') \in \texttt{S} \wedge \texttt{M}, \sigma \models \mathcal{A}ccess((,)\,o', \texttt{a}) \wedge Class(\texttt{o}')_{\sigma} \notin \{\texttt{Account}, \texttt{Bank}\}\,)\ \ ]$$

And by applying the Definition 1, again, we obtain that a module M satisfies **Pol_2**, if any configuration σ which arises from the combination of M with any other module M, also satisfies **Pol_2**. Formally:

[4]This part is as we had defined previously, with two simplifications: a) we do not need to worry about the **obeys**-predicate here, and b) we do not distinguish the names of the classes and the names of participants in interfaces.

[5]That is, $\mathcal{A}rising(\texttt{M}) = \{\ \sigma \mid \exists \texttt{e}.\ \texttt{M} \vdash (\texttt{e}, \emptyset) \rightsquigarrow *\sigma\ \}$

$$\mathtt{M} \models \mathbf{Pol\_2} \quad \longleftrightarrow \quad \forall \mathtt{M}'.\forall \sigma \in \mathcal{A}rising(\mathtt{M}' * \mathtt{M}).\ \mathtt{M}' * \mathtt{M}, \sigma \models \mathbf{Pol\_2}$$

7 REASONING ABOUT ENCAPSULATION

It is natural in programming to require that certain values do not "leak" out of data structures. For example, a `Account` does not leak an other `Account` or a `Node` or a `Bank`. [6]. Using the predicate we can specify that values are encapsulated.

We define a further policy

Pol_7 The `Bank` does not leak out of the `Bank`/`Account` system

And we give a formal specification

DEFINITION 4 (BANKS DO NOT LEAK). *We define* **Pol_7** *as follows:*

$$\mathbf{Pol\_7} \equiv \forall \mathtt{b}.\forall \mathtt{o}.\forall \mathtt{S}.\ [\ \ \mathtt{b} : \mathtt{Bank} \land \neg(\mathtt{o}' : \mathtt{Account}) \land \mathtt{b} \neq \mathtt{o} \land \mathcal{W}ill\mathcal{A}ccess\mathcal{T}hrough(\mathtt{o}, \mathtt{b}, \mathtt{S}) \longrightarrow \exists \mathtt{o}'.\ [\ \mathtt{o}' \in \mathtt{S}\ \land\ \mathcal{A}ccess((,)\,o', \mathtt{b})\ \land\ \neg(\mathtt{o}' : \mathtt{Account})\ ]\ \ ]$$

BIBLIOGRAPHY

[6] Such policies have not been required as such in [**?**], but are useful in reasoning about programs. TODO: Strengthen this discussion