Automated Verification Using Unified Control Flows

Cristian Gherghina Cristina David

Department of Computer Science, National University of Singapore
{*cristian,cristina*}*@comp.nus.edu.sg*

***Abstract*—Exception handling is an important feature in modern programing methodology. However, it introduces abnormal control flows due to which the analysis becomes considerable more difficult. In general, automated verification of programs with diversified control flows can prove to be a difficult problem. We propose an approach to verification that handles different control flow types in a unified manner.**

***Keywords*-Automated Verification; Exceptions; Separation Logic; Control Flow Unification**

I. Introduction

The verification system we propose is primarily concerned with capturing abnormal events, with the help of catch handlers for supporting recovery actions. Exceptions that are not being caught are usually considered as program errors, while those that are being caught and handled are assumed to be temporary conditions that could be rectified. Present-day type systems are rather poor at analysing exception-based programs, as they are either imprecise, or ignore exceptions altogether [3]. Though there are some studies that take into account exception handling ([1], [4]), they require a more complex formalization without even capturing the entire spectrum of control flow types. In the current work, we advocate for a verification system that can handle exceptions, program errors and other kinds of control flows through a uniform mechanism capturing static control flows (such as normal execution and multi-return calls) and dynamic control flows (such as exceptions) within a single formalism.

II. Control Flow Hierarchy

The key feature of our system is the control flow unification mechanism. For unifying the control flows, we rely on a subtyping relation, denoted by $<:$, defined over all control flows.

All control flow types are subtypes of `flow`. Control flows that can be caught are subtypes of `c-flow`, while `abort` denotes flows that can not be caught, program termination by halt or non-termination by hang. Although `hang` could be used to deal with non-termination, this capability is orthogonal to the focus of the current paper. For control flows corresponding to exceptions, the flow subtyping relation coincides with the class hierarchy having its root in the `exc` class, the super class for all exceptions.

Regarding the static control-flows, they are grouped under `local` and include `norm`, normal control flow, `ret_top` signaling the flow after a return from a method, flows generated by break, `brk_top` and continue, `cont_top`, within loop iterations.

A graphical representation of the entire flow hierarchy is given in Fig.1. Each arrow $c_2 \rightarrow c_1$ denotes a subtyping relation $c1 <: c2$.

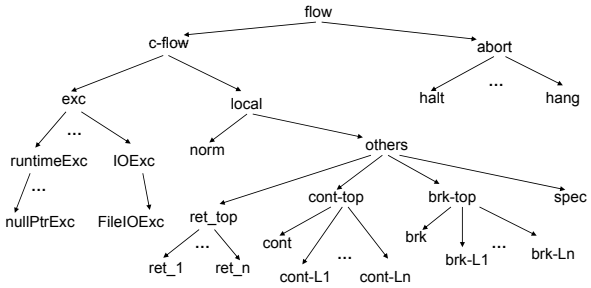

Figure 1. A Subtype Hierarchy on Control Flows

III. Input and Core Languages

In [2] we have designed a procedural core language, `Core−U` , on which we choose to perform the current verification. The reasoning behind our choice is that `Core−U` is considerable easier to verify than the input language, which is a fully fledged Java-like programming language. The translation from the input language to the `Core−U` language employs the rules described in [2]. The core language is syntactically minimal, whereas semantically it still maintains the expressivity of a full programming language.

In previous core languages with exceptions for Java, a variable `v` would return a value with normal flow, while `throw v` would invoke an exceptional flow based on the exception object in `v`. In our approach, we unify both these constructs with `ft#v`, whereby normal flow is realised by `norm#v` while exceptions may be thrown using `ty(v)#v`. The function `ty(v)` returns the runtime type of an exception object pointed by `v`.

Another major construct of our calculus is a try-catch mechanism of the form `try` $\mathtt{e}_1$ `catch` $((\mathtt{c@fv})\#\mathtt{v})$ $\mathtt{e}_2$ which specifies a control flow `c` and two bound variables to capture a control flow type `fv` and its thrown value `v`, provided that `fv`$<:$`c`. This try-catch construct is more general than that used in Java since it can capture not only

exceptional flow, but also normal flow and other abnormal control flows due to `break`, `continue` and `return` that can be translated to the corresponding control flows. As a pleasant surprise, the usual sequential composition $\texttt{e}_1;\texttt{e}_2$ is now a syntactic sugar for `try` $\texttt{e}_1$ `catch` (`norm#_`) $\texttt{e}_2$ whereby each _ denotes a distinct anonymous bound variable.

With regard to the verification process, each method must be annotated with pre and postconditions, and each loop must be provided a loop invariant. Given these specifications, our system will verify that the procedures actually satisfy the specifications. Each pre/postcondition is represented by a separation logic formula describing three major components, namely the shape of the allocated heap, the constraints over the flow variables and the arithmetic constraints over variables.

IV. Example

To illustrate our approach let us consider the following example highlighting the handling of exceptions. It consists of two methods, `withdraw` and `remove10`. The first method subtracts a sum from an account. If the balance is insufficient, then a `NoCred` exception is raised. The second method deducts 10 from the account. If the `withdraw` method raises a `NoCred` exception, `remove10` catches it and returns 0. Otherwise, it returns 1.

```
data acc{int number; int balance; }
class NoCred extends exc {}

void withdraw(acc a, int s) throws NoCred
  requires a::acc<i,b>
  ensures (a::acc<i,q>∧b=q+s∧b>s∧flow=norm)
 ∨(a::acc<i,b>∗res::NoCred<>∧b≤s∧flow=NoCred);
{ if (a.balance>s) a.balance = a.balance−s;
   else raise new NoCred();   }

int remove10 (acc a)
 requires a::acc<n1,v1>
 ensures ((a::acc<n1,v>∧v1>10∧v1=v+10∧res=1)
  ∨ (a::acc<n1,v1>∧v1≤10∧res=0))∧flow=norm;
{   try{
       withdraw(a,10);
    }catch (NoCred# v){return 0;};
    return 1;   }
```

Explicit flow constraints, `flow=NoCred`, can be included in the postconditions to indicate that the formula captures a state for which the flow is of type `NoCred`. By tracking the control flow types, our system is able to prove information that is unavailable to a type system. Namely, it is able to soundly check that the above methods meet their specifications.

V. Implementation

We have constructed a prototype system for verifying programs by making use of the above mechanism. The system relies on forward reasoning to compute the strongest program state for each program location. The computed states are used to prove the given postcondition.

We list below the key verification rules pertaining to control flow handling:

$$\frac{\begin{array}{c}[\textbf{FV-OUTPUT-DIRECT}]\\ \vdash \{\Delta\}\, ft\#v\, \{\Delta \wedge \texttt{flow} = ft \wedge \texttt{res} = v\}\\ [\textbf{FV-SPLIT}]\\ \Delta_1 = \exists \texttt{flow}, \texttt{res}.([\texttt{flow} \mapsto fv, \texttt{res} \mapsto v]\Delta \wedge fv = c)\\ \Delta_2 = \Delta \wedge \neg(\texttt{flow} = c)\end{array}}{\begin{array}{c}\mathit{split}(\Delta, c, fv, v) = (\Delta_1, \Delta_2)\\ [\textbf{FV-TRY-CATCH}]\\ \vdash \{\Delta\}\, e_1\, \{\Delta_1\} \quad (\Delta_2, \Delta_3) = \mathit{split}(\Delta_1, c, fv, v)\\ \vdash \{\Delta_2\}\, e_2\, \{\Delta_4\}\end{array}}$$

$$\vdash \{\Delta\}\, \texttt{try}\ e_1\ \texttt{catch}\ (c@fv\#v)\ e_2\, \{\exists v, fv \cdot (\Delta_3 \vee \Delta_4)\}$$

In the `Core−U` language there are two constructs that alter control flow types. The first one is `ft#v` which transforms a normal control flow into the flow type denoted by `ft`. The second one is the try/catch construct which can transform any flow type into a normal flow. The rule `FV−OUTPUT−DIRECT` describes how the system handles $ft\#v$. More specifically, the postcondition must ensure that the flow is of type ft, while the result of the execution has the given value v.

The try/catch construct permits an optional variable fv which is usually omitted. Its role is to capture the exact control flow handled by the catch clause, as opposed to the static type described by c in the rule `FV−TRY−CATCH`. Handling try/catch is a bit more challenging than the $ft\#v$ construct. After the body has been explored, the resulting formula can cover more than one program state and implicitly more than one control flow type. Therefore, we introduce a split operation that selects the subformula describing the states with a specific control flow type $fv{=}c$. On this subformula, the body of the catch is symbolically executed in order to obtain the strongest program state.

Take note that, apart from the above constructs, there is no other language construct requiring a special treatment with respect to control flow types.

Due to our unified view on control flow types, the try/catch construct can be unambiguously treated, as the subtyping hierarchy is always a tree instead of a graph. Moreover, the construct becomes more expressive as it allows the programmer to define behaviours for all subtypes of $c{-}flow$, not only for exceptional flows.

References

[1] Sophia Drossopoulou and Tanya Valkevych. Java exceptions throw no surprises. Technical report, March 2000.

[2] C. David and C. Gherghina and W. N. Chin Translation and Optimization for a Core Calculus with Exceptions In *PEPM*, 2009.

[3] Tobias Nipkow and David von Oheimb. Java_light is type-safe - definitely. In *POPL*, pages 161–170, 1998.

[4] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *FASE*, pages 284–303, 2000.