KeY-Style Verification for (Hybrid) ABS

Advances after KeY-ABS

Eduard Kamburjan
University of Oslo

KeYnote Series, 23.04.2021
Introduction

Programming Language
Dynamic Logic

Specification Language
External Tools

Publication:
Kamburjan, Scaletta, Rollshausen, Crowbar: Behavioral Symbolic Execution for Deductive Verification of Active Objects, abs/2102.10127
Available at https://github.com/Edkamb/crowbar-tool
Introduction

Object-Oriented
Sequential
Programming Language
Specification Language
Dynamic Logic
External Tools

Publication:
Kamburjan, Scaletta, Rollshausen
Crowbar: Behavioral Symbolic Execution for Deductive Verification
abs/2102.10127
Available at
https://github.com/Edkamb/crowbar-tool
Introduction
Introduction

Object-Oriented
Distributed / Hybrid
Programming Language

Dynamic Logic

Specification Language

External Tools

Publication:
Kamburjan, Scaletta, Rollshausen, Crowbar: Behavioral Symbolic Execution for Deductive Verification, abs/2102.10127
Available at https://github.com/Edkamb/crowbar-tool
Publication: Kamburjan, Scaletta, Rollshausen, Crowbar: Behavioral Symbolic Execution for Deductive Verification of Active Objects, abs/2102.10127

Available at https://github.com/Edkamb/crowbar-tool
Preliminaries
```java
class A(Int x, Bool locked, B other) implements A{
  Unit m(){
    Fut<Int> f = other!met(this.x);
    this.locked = True;
    await f?;
    this.locked = False;
    this.x = this.x + f.get;
  }
  Unit setX(Int x){ await !this.locked; this.x = x; }
}
```

- Object-private fields, interleavings only at `await`
- Object-Oriented Actors + Futures + Cooperative Scheduling
ABS

ABS is specifically designed to combine verification, analysis, execution and natural modeling (for programmers).

- Functional Language
  
  ```
  1 data IntList = Cons(Int, IntList) | Nil;
  2 def Int length(IntList l) = case l { ... };
  ```

- Symbolic time

  ```
  1 println(now()); //0
  2 duration(1,1);
  3 println(now()); //1
  ```

- ...

3/27
KeY-ABS

Developed 2015, keeps track of communication events during symbolic execution in a *history*. Trace properties are verified as object invariants over the history.

- FO logic over histories is not a good specification language
- Requires full symbolic execution to detect errors in the beginning of the method
- Implementation still retains Java-bindings:
  - Hard to connect with external tools
  - Hard to prototype new specifications
  - Hard to include functional sublanguage
### Behavioral Program Logic

#### Trace Properties

In a concurrent setting, (a) most properties of interest are trace-based and (b) no general scheme is established.

#### The Many Faces of the Box Modality for Traces

- \( [s] \forall i \in \mathbb{N}. \text{history}[i] \neq \text{invEv} \)  
  ABSDL [Din et al., SEFM'12]
Behavioral Program Logic

Trace Properties

In a concurrent setting, (a) most properties of interest are trace-based and (b) no general scheme is established.

The Many Faces of the Box Modality for Traces

- $[s] \forall i \in \mathbb{N}. \text{history}[i] \neq \text{invEv}$  
  ABDL [Din et al., SEFM’12]

- $[s] \Box \text{this}. f > 0$  
  DTL [Beckert and Bruns, CADE’13]

- [s] ☐ this.f > 0

And more...
**Trace Properties**

In a concurrent setting, (a) most properties of interest are trace-based and (b) no general scheme is established.

---

**The Many Faces of the Box Modality for Traces**

- $[s] \forall i \in \mathbb{N}. \text{history}[i] \neq \text{invEv}$  
  ASDL [Din et al., SEFM’12]

- $[s] \Box \text{this}.f > 0$  
  DTL [Beckert and Bruns, CADE’13]

- $[s] \text{finite} \ast \ast [\text{this}.f > 0] \ast \ast \text{finite}$  
  DLCT [Din et al., TABLEAUX’15&’17]
Behavioral Program Logic

Trace Properties

In a concurrent setting, (a) most properties of interest are trace-based and (b) no general scheme is established.

The Many Faces of the Box Modality for Traces

- \([s]\forall i \in \mathbb{N}.\text{history}[i] \neq \text{invEv}\)  
  ABSDL [Din et al., SEFM’12]

- \([s] \Box \text{this}.f>0\)  
  DTL [Beckert and Bruns, CADE’13]

- \([s]\text{finite} \ast \ast [\text{this}.f>0] \ast \ast \text{finite}\)  
  DLCT [Din et al., TABLEAUX’15&’17]

- \(s \vdash X!m(\text{this}.f>0).Y!n.\text{end}\)  
  Session Types for AO [Kamburjan and Chen, iFM’18]
Behavioral Program Logic

**Trace Properties**

In a concurrent setting, (a) most properties of interest are trace-based and (b) no general scheme is established.

**The Many Faces of the Box Modality for Traces**

- \([s] \forall i \in \mathbb{N}. \text{history}[i] \neq \text{invEv}\)
  - ASDL [Din et al., SEFM’12]
- \([s] \Box \text{this}. f > 0\)
  - DTL [Beckert and Bruns, CADE’13]
- \([s] \text{finite} \ast \ast [\text{this}. f > 0] \ast \ast \text{finite}\)
  - DLCT [Din et al., TABLEAUX’15&’17]
- \(s \vdash X!m \langle \text{this}. f > 0 \rangle . Y!n. \text{end}\)
  - Session Types for AO [Kamburjan and Chen, iFM’18]
- And more....
Behavioral Modalities

$\text{statement}$

$S$
Behavioral Modalities

\[ \begin{align*}
S & \quad \text{statement} \\
T & \quad \text{type}
\end{align*} \]
Behavioral Modalities

\[ \alpha \vdash \tau \]

semantics

statement  type
Example

Trace-specifications are too complex for simple post-conditions.
- ABSDL has object-invariant *implicit*
- BPL makes structure explicit
Example

Trace-specifications are too complex for simple post-conditions.

- ABSDL has object-invariant *implicit*
- BPL makes structure explicit

\[
\begin{align*}
\Gamma & \Rightarrow \{ U \} \{ x := v \} [ s \models ( \phi, \text{inv} ) ], \Delta \\
\Gamma & \Rightarrow \{ U \} [ x = v ; s \models ( \phi, \text{inv} ) ], \Delta
\end{align*}
\]
Example

Trace-specifications are too complex for simple post-conditions.

- ABSDL has object-invariant *implicit*
- BPL makes structure explicit

\[
\begin{align*}
\Gamma & \Rightarrow \{ U \} \{ x := v \}[s \vdash (\phi, inv)], \Delta \\
\Gamma & \Rightarrow \{ U \}[x = v; \ s \vdash (\phi, inv)], \Delta \\
\Gamma & \Rightarrow \{ U \} inv, \Delta \\
\Gamma, \{ U_A \} inv & \Rightarrow \{ U_A \}[s \vdash (\phi, inv)], \Delta \\
\Gamma & \Rightarrow \{ U \}[\text{await } e?; \ s \vdash (\phi, inv)], \Delta \\
\end{align*}
\]
Crowbar
Behavioral Symbolic Execution

Crowbar is a symbolic execution engine to prototype behavioral symbolic execution: SE influenced by its context.

Aims

- Investigate how SE can cooperate with rest of static toolchain
- Quicker development cycles than KeY/Java
**Frontend: Specification**

<table>
<thead>
<tr>
<th>Supported Specification Approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Cooperative method contracts (with <code>\old</code> and <code>\last</code>)</td>
</tr>
<tr>
<td>• Object invariants</td>
</tr>
<tr>
<td>• Session Types</td>
</tr>
</tbody>
</table>

```plaintext
Spec: LoopInv(i>=0)
while (i > 0) i = i - 1;
```
Frontend: Specification

Supported Specification Approaches

- Cooperative method contracts (with \old and \last)
- Object invariants
- Session Types

- Only user-input is a complete ABS program to integrate with the parser and type system.
- Specifications are annotated directly in the program.

```
1 ...
2 [Spec: LoopInv(i>=0)]
3 while(i > 0) i = i-1;
4 ...
```
Nullability Guides

Nullability Types

Most null-pointer exceptions can be handled by the type system. ABS has a lightweight analysis to mark expression as non-null.

```java
Unit m([NonNull] C o, C o2){
  Int i = o.m(); //safe
  Int j = o2.m();
  Int k = o2.m(); //safe
  return i + j + k;
}
```

- Crowbar keeps this information in the AST
- Safe accesses do not cause branching
Step 1: Generating local types for objects

Step 2: Propagating guarantees in objects

Step 3: Generating local types for methods
Step 1: Generating local types for objects

Step 2: Propagating guarantees in objects

Step 3: Generating local types for methods

• Propagation is outside Crowbar

• Each class generates a static node for projection
Step 1: Generating local types for objects
Step 2: Propagating guarantees in objects
Top-Down Specification with Session Types

Step 1: Generating local types for objects

Step 2: Propagating guarantees in objects

Step 3: Generating local types for methods
Top-Down Specification with Session Types

Step 1: Generating local types for objects

Step 2: Propagating guarantees in objects

Step 3: Generating local types for methods

- Propagation is outside Crowbar
- Each class generates a static node for projection
Session Types

class C(Server s, Client c, Database d) {
    [Spec:Local("db!reset().(server!m(a > 3))*.Put()")]
    Unit sideconditionInLoop() {
        Fut<Int> sth = this.d!reset();
        Int a = 10;
        while(a > 5) sth = this.s!m(a--);
    }
}

\[
\Gamma \Rightarrow \{U\}(x = this.f \land \phi), \Delta \quad \Gamma \Rightarrow \{U\}\{v := f\}[s \models L], \Delta
\]

\[
\Gamma \Rightarrow \{U\}[v = this.f!m(); s \models x!m(\phi).L], \Delta
\]
ABS has a functional sublanguage for ADTs. Each definition is translated into an assignment with contracts.

1. [Spec: Requires(n >= 0)] [Spec: Ensures(result >= 0)]
2. def Int fac(Int n) = if(n<=1) then 1 else n*fac(n-1);

1. [Spec: Requires(n >= 0)] [Spec: Ensures(result >= 0)]
2. Int fac(Int n){
3. \quad return if(n<=1) then 1 else n*this.fact(n-1);
4. }
Functional Layer

Functions and Data in ABS

ABS has a functional sublanguage for ADTs. Each definition is translated into an assignment with contracts.

1 \[\text{Spec: Requires}(n \geq 0)\] \[\text{Spec: Ensures}(result \geq 0)\]

2 \textbf{def} \textbf{Int} \textbf{fac}(\textbf{Int} \ n) = \textbf{if}(n \leq 1) \textbf{then} 1 \textbf{else} n \ast \textbf{fac}(n-1);

\[
(\forall \ \textbf{Int} \ x. \ x \geq 0 \rightarrow \textbf{fac}(x) \geq 0) \land n \geq 0
\]

\[
\rightarrow [\text{result} = \textbf{if}(n \leq 1) \textbf{then} 1 \textbf{else} n \ast \textbf{fac}(n-1); \ |-^{\alpha_{\text{pst}}} \text{result} \geq 0]\]

- ABS does not support first-order function passing
- ADTs are translated into SMT-LIB datatypes
Crowbar implements symbolic execution with **guides**: additional inputs that guide execution and shape the symbolic execution tree.

**Rules**

- Rules Kotlin classes implementing

```
abstract class Rule(val conclusion : Modality) {
  abstract fun transform(cond : MatchCondition,
                          input : SymbolicState): List<SymbolicTree>
}
```

- Matching is implemented directly on the AST using reflection: Schemavariables are any instances implements `AbstractVar`
User Feedback

While non-interactive, Crowbar must still give comprehensive feedback to user and developer. We generate a program from failing proof branch and annotate relation to specification.
Experiences with Crowbar
Experiences with Crowbar

**C2ABS** [Wasser et al., SCP’21]

Translates ACSL-specified C-Code into ABS.
Underspecified semantics becomes non-deterministic concurrency.

**Example**

Following code returns 1 (clang) or 2 (gcc)

```c
int x;
int id_set_x(int val){
    x=1; return val;
}
int main(void){
    x=0; return x + id_set_x(1);
}
```
Experiences with Crowbar

Case Study
Highly underspecified variant of $\text{fib}(n)$ which returns a number between 1 and the $n$th Fibonacci number based on evaluation order.

- 4 C functions, each with post-conditions, 1 Strong invariant
- Translation generates 260 lines of ABS code
  - 5 classes (with invariants and creation conditions)
  - 5 interfaces with 19 method contracts
  - 1 function with contract

Old KeY-ABS case study: 140 LoC, 1 class, 1 invariant, interactive
### Advances in Language Coverage over KeY-ABS

- Covers complete imperative layer of CoreABS without exception handlers
- Covers functional layer without `let`
- Specification integrated into ABS

### Missing Pieces

- Explicit history using the functional layer and ghost statements
- First-Order Specification and full ABS Session Types
- Additional backends (Why3, KeY-Java, ...)
- Restarting SE for further modalities
Hybrid ABS
Distributed Cyber-Physical Systems

Many modern systems are distributed CPS with isolated dynamics: IoT, Industrie 4.0, Digital Twins, ...

How to (a) model (b) simulate and (c) verify such systems?

Hybrid ABS

- Modeling: Hybrid ABS = ABS + ODEs.
- Verifying: Crowbar + KeYmaera X
class CSingleTank(Real inVal) {
    physical {
        Real lvl = inVal : lvl' = flow;
        Real flow = -0.5 : flow' = 0;
    }

    Unit run() { this!up(); this!low(); }

    Unit low() {
        await diff lvl <= 3 & flow <= 0;
        flow = 0.5; this!low();
    }

    Unit up() {...}
}
Object Invariants

Proof Obligations with Dynamic Logic

In discrete systems, an object invariant $I$ can be checked *modularly* with dynamic logic by showing that every method preserves $I$.

$$I \rightarrow [s]I$$  

Proof Obligation for Java

This uses that the state does not change in inactive objects.
In discrete systems, an object invariant $I$ can be checked *modularly* with dynamic logic by showing that every method preserves $I$.

$$I \rightarrow [s]I$$  

**Proof Obligation for Java**

This uses that the state does not change in inactive objects.
In discrete systems, an object invariant $I$ can be checked \textit{modularly} with dynamic logic by showing that every method preserves $I$.

$$I \rightarrow [s]I$$

\text{Proof Obligation for Java}

This uses that the state does not change in inactive objects.
Object Invariants

Challenge: Express that \( I \) holds until the next method runs.
Challenge: Express that $I$ holds until the next method runs.

Differential Dynamic Logic

A logic for (algebraic) hybrid programs:

$$\phi ::= \forall x. \phi \mid \ldots \mid [\alpha] \phi \quad \alpha ::= ?\phi \mid v := t \mid \{ v' = f(v) & \phi \} \mid \ldots$$

Example

Set a variable to 0, let it raise with slope 1 while it is below 5 and discard all runs where it is above 5.

$$\left[ x := 0 ; \{ x' = 1 \& x \leq 5 \} \right] x \geq 5$$

This formula is valid.
Object Invariants

Challenge: Express that I holds until the next method runs.

Differential Dynamic Logic

A logic for (algebraic) hybrid programs:

\[ \phi ::= \forall x. \phi \mid \ldots \mid [\alpha] \phi \quad \alpha ::= ?\phi \mid v := t \mid \{v' = f(v) & \phi\} \mid \ldots \]

Example

Set a variable to 0, let it raise with slope 1 while it is below 5 and discard all runs where it is above 5.

\[ [x := 0; \{x' = 1 & x \leq 5\}; ?x \geq 5] x \div 5 \]

This formula is valid.
Proof Obligations for Hybrid Active Objects

If we can translate the method body of a method into $d\mathcal{L}$, then we can express that the invariant holds once the method ends and keeps holding when following the dynamics.

$$I \rightarrow \left[ s \right] \left( I \land [\text{ode}&\text{true}] I \right)$$

Proof Obligation for HABS

Using true means that $I$ must hold forever...
Controlled Regions

Let $g$ be disjunction of the guards of the methods that are known to be in the scheduler queue after a method terminates. To establish $I$, the following proof obligation suffices:

$$I \rightarrow [s] (I \land [ode \& \neg g] I)$$

Example

1. Real m() { ... this!other(); return 0; }
2. Real other() { await diff x >= 0; ... }

$$I \rightarrow [\ldots] (I \land [ode \& x \leq 0] I)$$
class CSimpleSingleTank(Real inVal) {

    physical{
        Real lvl = inVal : lvl' = flow;
        Real flow = -0.5 : flow' = 0;
    }

    Unit run() { this!up(); this!low(); }
    Unit low() { await diff lvl <= 3; flow = 0.5; this!low(); }
    Unit up() { await diff lvl >= 10; flow = -0.5; this!up(); }
}

I → [...]((I ∧ [ode&lvl ≥ 3 ∧ lvl ≤ 10]) /)
Chisel \cite[Kamburjan, HSCC’21]{Kamburjan2021}

Post-Regions are implemented as a translation into KeYmaera X.

- Also supports method contracts and local Zeno Behavior.
- Interoperable with Crowbar through method contracts.
- Only supports \texttt{Real} variables
Post-Regions are implemented as a translation into KeYmaera X.

- Also supports method contracts and local Zeno Behavior.
- Interoperable with Crowbar through method contracts.
- Only supports Real variables

**Future Work**

Is it possible to move all ODEs out of the first program?

```plaintext
duration(5); I → [s; {ode&t ≤ 5}; s'](I ∧ [ode&true]I)
```
Crowbar: A flexible framework for prototyping deductive verification of distributed object-oriented programs.
Crowbar: A flexible framework for prototyping deductive verification of distributed object-oriented programs.

### On-Going and Future Work

- Redo the KeY-ABS case studies in Crowbar
- Rules as Kotlin DSL
- Comparison of trace specifications/logics in Crowbar
- Verification of coupled objects in Hybrid ABS
Conclusion

Crowbar: A flexible framework for prototyping deductive verification of distributed object-oriented programs.

On-Going and Future Work

- Redo the KeY-ABS case studies in Crowbar
- Rules as Kotlin DSL
- Comparison of trace specifications/logics in Crowbar
- Verification of coupled objects in Hybrid ABS

Long-term goal

Reintegration with KeY as a KeY-ABS successor
Conclusion

Crowbar: A flexible framework for prototyping deductive verification of distributed object-oriented programs.

On-Going and Future Work

- Redo the KeY-ABS case studies in Crowbar
- Rules as Kotlin DSL
- Comparison of trace specifications/logics in Crowbar
- Verification of coupled objects in Hybrid ABS

Long-term goal

Reintegration with KeY as a KeY-ABS successor

Thank you for your attention