Deductive Verification of Active Objects with `Crowbar`

Eduard Kamburjan[a], Marco Scaletta[b], Nils Rollshausen[b]

[a] *University of Oslo, Oslo, Norway*
[b] *Technische Universität Darmstadt, Darmstadt, Germany*

Abstract

We present `Crowbar`, a deductive verification tool for the Active Object language `ABS`. `Crowbar` implements novel specification approaches specifically for distributed systems. For user interaction, counterexamples are presented as executable programs. `Crowbar` has a modular structure to explore further approaches, and was applied in the largest Active Objects verification study.

Keywords: Deductive Verification, Symbolic Execution, Active Objects

1. Introduction

Deductive verification of functional properties is a static analysis technique that uses *program logics* to verify the behavior of programs against user-provided specification. Provers implementing such program logics, for example the `KeY` [1] prover, have been successfully used to detect bugs in well-tested libraries [2, 3] for sequential programs. `KeY` implements *heavyweight symbolic execution (SE)*, where the program is transformed into first-order formulas by keeping track of symbolic values throughout program execution. Heavyweight symbolic execution is able to deal with unbounded systems by additional user-provided specification, for example through additional invariants that have to be preserved by loops.

For distributed systems, more complex specification languages than for sequential programs are needed. The specific concurrency model plays a major role in the design of specification languages and program logics, and in this work we present a heavyweight symbolic execution tool for Active Objects.

Email addresses: `eduard@ifi.uio.no` (Eduard Kamburjan), `scaletta@cs.tu-darmstadt.de` (Marco Scaletta), `nils.rollshausen@stud.tu-darmstadt.de` (Nils Rollshausen)

Active Objects [4] are an object-oriented, actor-based concurrency model for distributed systems, developed specifically with a focus on analyzability and implemented in the `ABS` language [5]. `ABS` has been applied as a modeling tool to a multitude of domains, ranging from cloud systems [6, 7, 8, 9], over railway operations [10] and memory systems [11] to computational biology [12].

A prototypical extension of `KeY`, `KeY-ABS` [13], showed that the general ideas of heavyweight symbolic execution indeed carry over to `ABS` [14]. However, new specification approaches [15, 16, 17], program logics [18, 19], integration with static analyses [20] and further case studies [21] have revealed that the structure of `KeY` with its tight integration of Java-specifics is not suited to handle (a) the specification languages needed for Active Objects and (b) parts of the `ABS` language that clash with Java specifics. For example, the functional sublanguage of `ABS`, exceptions and method contracts are not supported. Consequently, `KeY-ABS` supports only object invariants and a small subset of `ABS` – namely the one which has similar semantics to Java.

`Crowbar` is the an alternative system, implemented from scratch and based on the Behavioral Program Logic (BPL) [18]. It covers full `coreABS`[1], delegates part of the proof obligation to external static analyses and uses the newly developed specification approaches for Active Objects, such as cooperative contracts [17]. `Crowbar` is not interactive, i.e., the user cannot see the proof in the program logic. To realize feedback, it instead integrates a counterexample generation that presents failed proof branches completely in terms of the program: the counterexample is output as an `ABS` program that is executable and contains only the statements needed to reproduce a run described by the failed proof branch. Thus, the user is not exposed to the underlying program logic. To accommodate the need for quick prototyping of new specification approaches, `Crowbar` has a modular structure suited for prototypical exploration of novel approaches: it is easy to add new BPL calculi. `Crowbar` was used for the biggest Active Object verification case study [22], that goes beyond the capabilities of prior systems in (1) language coverage, (2) complexity of specification, and (3) lines of code.

Targeted Problem.. `Crowbar` can verify functional properties of Active Objects specified by cooperative method contracts [17] and local session types for Active Objects [15], both formalized in the Behavioral Program Logic [18]. It implements delegating parts of the proof obligation to external static analy-

[1] `ABS` without its extensions for, e.g., timed models.

ses and has a modular structure that enables easy extendability to implement further behavioral specifications, which we describe in more details below.

Structure.. This article is structured as follows. Sec. 2 introduces Active Objects, `ABS`, the used specification languages, symbolic execution and gives an example. Sec. 3 describes the structure of `Crowbar`. Sec. 4 describes the implementation in more detail, the aforementioned case study and compares `Crowbar` with `KeY-ABS` and other related tools. Sec. 5 gives an example of the usage of `Crowbar` and the counterexample generator before Sec. 6 concludes.

For the technical documentation we refer to the documentation of `Crowbar` at `https://github.com/Edkamb/crowbar-tool/wiki`, the theoretical background is referred to in the corresponding parts of Sec. 2.

2. Background

To verify the safety of a program one must specify its expected behavior and translate the specified program into a set of *proof obligations*. If all proof obligations can be discharged, then the program is *safe*, where under safety we understand (local) partial correctness [23]: if every process terminates, then the program behaves as specified and no process throws an exception.

For distributed systems, with their inherit non-determinism, it is crucial that the specification is *modular*. This means that changes in one part of the program should not invalidate all proof obligations. Thus, the design of specification techniques is a balancing act between strong abstraction with high modularity and weak abstraction with high expressive power [5]. `ABS` and `Crowbar` are specifically designed to explore this trade-off by integrating different approaches in one system to compare them in practice. `Crowbar` is also modular in its approach to integration and allows to interact with type systems and static analyses.

2.1. Active Objects and ABS

Active Objects are an actor-based, object-oriented concurrency model which realizes strong encapsulation. At its core, an Active Object program consists of a set of objects, which communicate with each other using asynchronous method calls, and futures to retrieve the return value of a method call. In each object, at most one process is active at any time, meaning that there is no interleaving within an objects. The objects are preemption-free:

a process must explicitly deschedule itself before another can become active. Objects can be created at runtime.

In more details, it is based on the following features for concurrency:

Strong Encapsulation. Every object is strongly encapsulated at runtime, such that no other object can access its fields, not even objects of the same class.

Asynchronous Calls with Futures. Each method call to another object is asynchronous and generates a future. Futures can be passed around and are used to synchronize on the process generated by the call. Once the called process terminates, its future is *resolved* and the return value can be retrieved. We say that the process *computes* its future.

Cooperative Scheduling. At every point in time, at most one process is active per object and a running process cannot be interrupted unless it *explicitly* releases the object. This is done either by termination with a **return** statement or with an **await** `g` statement that waits until guard `g` holds. A guard polls whether a future is resolved or whether a boolean condition holds.

Concurrent systems are challenging for SE, but Active Objects allow to perform *local* SE on single methods through their strong encapsulation and decoupling of caller and callee processes. Special care, however, has to be taken to keep track of futures and correct handling of state when using **await**. We introduce the `ABS` language using an example to demonstrate these features.

ABS. The Abstract Behavioral Specification language (`ABS`) is an implementation of Active Objects with a rich toolkit of static analyses. It supports extensions of Active Objects such as product lines [24], timed [25], hybrid [26], resource [27] models or open systems [28]. The fragment which only implements the core Active Objects features is referred to as `coreABS`. `Crowbar` only supports `coreABS`[2]. Beyond the object-oriented language, `ABS` has a simple (no function passing) functional sublanguage for data processing. For a detailed tutorial we refer to the online material of `ABS`[3].

[2] It also supports product lines in the sense that after a variant is generated, `Crowbar` can verify it. It does not perform a family-based analysis [29].

[3] http://abs-models.org

2.2. Specification

Specification of Active Objects must take concurrency into account, even method contracts require special attention due to, among others, the time gap between calling a method and starting its execution. For `Crowbar` we support state object invariants, behavioral method contracts and local session types. `Crowbar` specifications are part of the input `ABS` file using `Spec`-*annotations*: Each ABS statement and definition `s` can be prefixed with an annotation using the `[T: e]s` syntax, where `T` is a type and `e` an expression. `Spec` defines the data type for specifications, which must be provided as the expression of the annotation. `Spec`-annotations are ignored at runtime. Additionally, loops are annotated with loop invariants using `WhileInv`.

Behavioral Method Contracts. A method can be annotated with `Ensures` and `Requires` as post and preconditions. In interfaces, these specifications can only contain parameters (and `result`, the special variable for the return value). For an example, we refer to Fig. 1.

Following the principles behind the Java Modeling Language (JML) [30], `Crowbar` supports `old` to refer to the pre-state of the method and `last` to refer to the pre-state of the last suspension, i.e., the state before the last **await** statement was executed, or the pre-state of the method if no **await** was executed yet. Additionally, `Crowbar` supports the post and preconditions at **await** suspension statements, as well as `Succeeds` and `Overlaps` context sets [17]: if the precondition of a method contains assertions about the heap, then it is not clear which method is responsible to establish it — due to the concurrency model, the caller has no control over the fields of the callee object. Context sets specify for a method `m` the following: the methods in `Succeeds` *must* have run and *must establish* the precondition as their post-condition. The methods in `Overlaps` *may* have run and *must preserve* the precondition.

Object Invariants. Objects are annotated with creation conditions (`Requires`) and object invariants (`ObjInv`). A creation condition describes the parameters of the constructor (analogously to asynchronous contracts), while the object invariant has to hold after the constructor terminate and whenever a process is scheduled or descheduled.

Local Session Types. Lastly, `Crowbar` supports a variant of local session types. A local session type for `ABS` is represented as a string and specifies calls, synchronization, sequence, repetition and alternative. Alternative is specified using `+`, repetition with `*`, suspension with `Susp`(φ) (where φ specifies the state before suspension, and analogously for calls `!`) and synchronization with `Get(e)`, where `e` is the targeted expression. Session types specify how *roles* in a protocol communicate. The mapping of roles to fields is specified with `[Spec: Role("name",this.field)]`. For details we refer to [18, 31, 15].

For example, the following expresses that the statement `s` first calls method `m` on role `f` and then `n` on role `g`. Finally, the last action is a **return** in a state where `result == 0` holds (i.e., the return value is zero). No other communication, synchronization or suspension happens. This is annotated using `[Spec: Local("f!m.g!n.Put(result == 0)")]`. The systems are independent: it is not necessary to specify a local type.

2.3. Examples

Example Contracts and Invariants. We give a short example on ABS and its specification below. For brevity's sake, we only give asynchronous method contracts and the object invariant.

Consider Lst. 1. The class `Monitor` has two fields: `s` which is a server that is monitored and `beats`, a counter for successful requests to `s`. The method `heartbeat` sends a request to itself (l. 7) by an asynchronous method call (by using `!`). Afterwards, the return value of the call is retrieved (l. 8, using **get**). This blocks the process until the `httpRequest` process has terminated. No other process can run on this object until this happens. If the request was successful, `beats` is increased by 1. Method `reset` waits *without blocking* until the number of success reaches a passed threshold and resets `beats`. Synchronous calls are possible (l. 10) on **this**.

Specifications are annotations of the form `[Spec: KIND(e)]`, where `KIND` is the used specification pattern. Fig. 1 gives an overview over the keywords for the specification patterns and where to annotate them. In the example, a creation condition and an object invariant are specified for the class `Monitor`. They state that the passed server must be non-null and stays non-null throughout execution (l. 1). The method contract states that the `beats` field is increased. This is not the case, and we return to the fault in Sec. 5.

Example Local Session Types. Fig. 2 shows an example how the specifications work together when using local session types. The class `C` uses a protocol with three roles, that are declared in the class header using `Role`. Additionally, an object invariant is used to declare that all the fields are non-null. The

```
[Spec:Requires(this.s != null)][Spec:ObjInv(this.s != null)]
class Monitor(Server s) {
 Int beats = 0;
 [Spec:Ensures(this.beats >= old(this.beats) &&
                result == this.beats)]
 Int heartbeat() {
  Fut<Int> req = s!httpRequest();
  Int status = req.get;
  if(status == 200) { this.beats = this.beats + 1;}
  else { this.handleError(); }
  return this.beats;
 }
 Unit reset(Int i){ await this.beats == i; this.beats = 0; }
 Unit handleError() { this.beats = 0; /* ... */ }
}
```

Listing 1: An example `ABS` program with specification.

sole shown method, `getExpLocalAliasing`, is using a type where as the first action the `client` is called on method `a`. The connection of the field **this**.`c` and the role `client` is established through the aforementioned class-level specification. The invariant is needed to show that no exception is thrown. The other actions specify a synchronization on the future stored in variable `f` and a termination without a specific post-condition.

2.4. Symbolic Execution

Symbolic execution describes the execution of a program (or statement) with *symbolic values*. A symbolic value is a placeholder and can be described by condition collecting during the symbolic execution. Heavyweight symbolic execution is used as a *proof strategy* for a sequent calculus of first-order dynamic logics in, e.g., JavaDL [1], ABSDL [32] or DTL [33] and has successfully applied to discover highly involved bugs in non-concurrent libraries of mainstream languages [2, 3]. One of the shortcomings of symbolic execution with dynamic logics is that they first fully symbolically execute the program and then evaluate the post-condition. For distributed systems the specification, however, often contains a temporal element and can be partially checked already *during* symbolic execution.

The Behavioral Program Logic (BPL) [18] is a generalization of dynamic

```
[Spec: Role("server", this.s)][Spec: Role("client", this.c)]
[Spec: Role("database", this.d)]
[Spec: ObjInv(this.s != null && this.c != null && this.d != null)]
class C(Server s, Client c, Database d) {
  [Spec:Local("client!a(true).Get(f).Put(true)")]
  Unit getExpLocalAliasing() {
    Fut<Int> f = this.c!a();
    Fut<Int> sth = f;
    Int a = sth.get;
  }
...
}
```

Figure 2: An example `ABS` program with a local session type.

logic. It uses *behavioral* modalities, which we informally introduce now, to enable such symbolic execution strategies. BPL is based on behavioral specifications $\mathbb{T} = (\alpha_{\mathbb{T}}, \tau_{\mathbb{T}})$. Set $\tau_{\mathbb{T}}$ is the set of terms of the specification, and function $\alpha_{\mathbb{T}}$ provides the semantics: it maps elements of $\tau_{\mathbb{T}}$ to trace formulas. Behavioral specification are referred to in the logic using behavioral modalities, which have the form $[\mathtt{s} \Vdash^{\alpha_{\mathbb{T}}} \tau]$, with $\tau \in \tau_{\mathbb{T}}$ being a specification term. Its semantics expresses partial correctness: a state σ satisfies the modality, if every trace θ of a normally terminating run of $\mathtt{s}$ from σ is a model for the trace formula $\alpha_{\mathbb{T}}(\tau)$. We omit α from examples for brevity.

A sequent has the form $\Gamma \Rightarrow \{U\}[\mathtt{s} \Vdash^{\alpha} \tau], \Delta$, where Γ and Δ are two sets of formulas, and represents a symbolic state, where $\mathtt{s}$ is the statement left to symbolically execute, U is the state update (i.e., a syntactic representation of accumulated substitutions [34]), and τ is the specification term, e.g., the post-condition, and $\bigwedge \Gamma \wedge \neg \bigvee \Delta$ describes the accumulated knowledge and path condition. An example rule for symbolic execution in sequent calculi is the following rule, that expresses a split over the branches of a branching statement for post-conditions φ.

$$\frac{\Gamma, \{U\}\mathtt{e} \Rightarrow \{U\}[\mathtt{s}_1 \Vdash \varphi], \Delta \qquad \Gamma, \{U\}\neg\mathtt{e} \Rightarrow \{U\}[\mathtt{s}_2 \Vdash \varphi], \Delta}{\Gamma \Rightarrow \{U\}[\mathbf{if}(\mathtt{e})\ \mathtt{s}_1\ \mathbf{else}\ \mathtt{s}_2 \Vdash \varphi], \Delta}$$

Behavioral specifications separate *syntax* of the specification (τ) and its semantics as a trace specification (α). This distinction enables behavioral symbolic execution: the *design* of τ can now aim to have a simple proof calculus that is not restricted by the structure of the logic underlying α. Furthermore, τ can serve as an interface to external analyses.

Established calculi for dynamic logics perform symbolic execution by reducing the statement inside a modality without considering the specification. In contrast, for each step in a behavioral type system [35], the statement is matched with the current specification. BPL combines both: a logical framework with a behavioral type-style calculus, which we call *guided SE*.

For example, the local session type given above is expressed with the modality $\left[\mathtt{s} \Vdash \mathtt{f!m.g!n.} \downarrow \mathtt{result == 0}\right]$. One (slightly prettified) rule for session types is the following. The first premise checks that the role and field coincide, the second premise *reduces the type* during the symbolic execution step. Note that this premise contains no modality – we call such branches *side-branches*. We stress that the conclusion of the rule has to match (1) the call in specification and statement and (2) the method names both *syntactically*. If matching fails, SE stops.

$$\frac{\Gamma \Rightarrow \{U\}\mathbf{this}.\mathtt{f} \doteq \mathtt{r}, \Delta \qquad \Gamma \Rightarrow \{U\}\left[\mathtt{s} \Vdash \mathtt{L}\right], \Delta}{\Gamma \Rightarrow \{U\}\left[\mathbf{this}.\mathtt{f!m();s} \Vdash \mathtt{r!m.L}\right], \Delta}$$

3. Software Framework

At its core, Crowbar is a *heavyweight* symbolic execution (SE) engine, i.e., it uses contracts and loop invariants to build a SE tree that abstracts from *all* possible runs. As discussed the used program logic allows for *guided* SE: the specification is used to guide the construction of the SE tree. Construction may abort if there is not possible that any further execution may satisfy the specification. For example, if the specification expresses that the first interaction is a call to a method m, but the first call is to a method n, then guided SE will immediately abort the proof. The leaves of the SE tree, after SE successfully finished, are logical formulas that are passed to SMT-solvers.

Fig. 3 illustrates guided SE using our example for Session Types. The node marked with (1) is not generated, as the type system ensures that **this**.f is always non-**null** (due to being annotated with NonNull). The other dashed nodes are omitted because the method is specified to make two calls, but executes three. It does not check any steps after the second call as these are already following a wrong execution path. We omitted (a) the update, (b) the collected path condition and (c) the role check branches in the illustration.

```
class C([NonNull]I f){
  I g;
  [Spec: f!m.g!n. ↓ φ]
  Unit m(){
    this.f!m();
    this.g!m();
    this.f!m();
    return 0;
  }
}
```

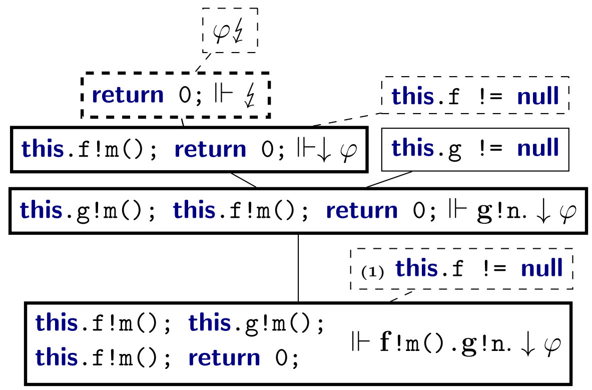

Figure 3: Illustration of guided SE. Verification of the method to the left results in a symbolic execution tree where the dashed nodes are not generated.

3.1. Software Functionalities

`Crowbar` implements the three specification paradigms described in Sec. 2 and the **assert** statement for `ABS`: *object invariants* [36], predicates that have to hold at every point a process gains or loses control over its object, *co-operative contracts* [17], a generalization of method contracts to distributed systems, and *local Session Types* [37, 16, 15], a protocol language for allowed communication actions.

It generates proof obligations in BPL and has two additional mechanisms to interact with the outside: (1) *Counterexample Generation:* If a proof obligation fails, all failed proof branches are translate back into an `ABS` program, using the values extracted from the SMT solver proof attempt at the leaf. This allows the user to investigate the failure without being exposed to the underlying program logic. (2) *Static Nodes:* Cooperative Contracts and Session Types rely on additional mechanisms to guarantee safety of composition (propagation for contracts and projection for Session Types). These mechanisms are external to the program logic and `Crowbar`, thus, outputs a *static node* to communicate that the program is safe, if these external conditions hold.

A verification attempt with `Crowbar` outputs either (1) yes, (2) yes with external condition or (3) no with counterexample (if generation succeeds).

`Crowbar` supports pre-/post-conditions for the functional sublanguage. `Crowbar` also integrates results directly from the `ABS` compiler: the AST has nullability annotations, and expressions are not checked for **null**-access if the type system already ensures this.

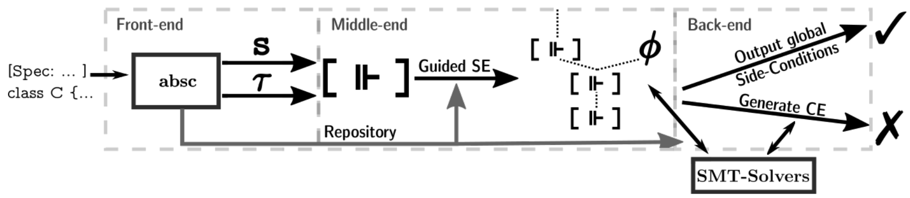

Figure 4: Structure of `Crowbar`.

3.2. Software Architecture

`Crowbar` has a pipeline setup with front-end, middle-end and back-end, as shown in Fig. 4.

Front-end. The front-end uses the `ABS` parser to generate an annotated AST and extracts, per method, one statement `s` and one specification term τ for this method. The statement is translated into an internal representation (IR) to normalize the AST. For example, each call has a target variable. The specification term language depends on the specification mechanism chosen by the user. The front-end furthermore sets up a program repository to manage the specification and connects the IR with the original.

Middle-end. The middle-end implements SE using the chosen set of rules. When SE finishes, all leaves of the SE trees are either *static nodes*, discussed above, *logical nodes*, which are first-order formulas whose validity ensures that this branch is safe. Guided SE is realized by matching the current specification term on the current program and reducing *both* in one SE step.

Back-end. The back-end of `Crowbar` passes logical nodes to external SMT-LIB solvers. If all logical nodes can be proven to be valid, then the program is considered safe up to external restrictions, which are output as static nodes. If one of the logical nodes fails to be proven valid, `Crowbar` attempts to construct a *counterexample program* [38] by extracting value from the counterexample model output by the SMT solver and reconstructing a minimal runnable program directly from the path of the SE tree taken to this leaf. If a SMT model is not available, counterexample generation fails. Generating the SMT-LIB input requires the repository to ensure correct typing.

4. Implementation and Empirical Results

Implementation. `Crowbar` is implemented in Kotlin in 5500 lines of SLoC (according to `cloc` [39]), and a ANTLRv4 parser for local Session Types. The build system is using gradle, the used testing framework is kotest. User manual and developer documentation on adding a new specification/verification module is available in the github repository. We performed two evaluations: For a performance evaluation, we use the `absexamples`[4] repository, where case studies and examples from the development of ABS are collected. We have loaded all 647 methods which are fall into the CoreABS fragment and used Crowbar to verify the default specification. As the default specification is not meaningful, we additionally adopted two bigger examples to CoreABS (WaterTank.abs, 60 LoC, chat.abs, 307 LoC), specified and verified that no exceptions are thrown (and a simple functional invariant of the water tank: water level never drops below 0). Benchmarking was run on a 8-core i7-8565U CPU with 1.80GHz and 32GB RAM on a Ubuntu 20.04.5 laptop. On average, 10 symbolic execution steps are performed (with 171 symbolic execution steps for the biggest), which on average needs $\sim 110ms$ (with $1650ms$).

As `Crowbar` requires a fully specified program to return meaningful results, we performed the following case study for functional correctness for a qualitative evaluation.

Case Study. `Crowbar` is used in the biggest verification case study for Active Objects [22]: A model extracted from C code [40] with 260 lines of `ABS` code, with 5 classes (with invariants and creation conditions), 5 interfaces (with 19 method contracts) and 1 function with a contract. The verification succeeds fully automatically. In contrast, the previously biggest case study [14] has 140 LoC for 1 class (with invariant) and requires manual interaction.

Coverage and Comparison with `KeY-ABS`. `Crowbar` covers full `coreABS`, i.e., `ABS` without its extensions for time or variability. Specification in `Crowbar` is purely in terms of the program, i.e., using expressions, using the specification patterns described in Sec. 2.

Table 1 gives the syntax for specifications and compares `Crowbar` and `KeY-ABS` with respect to language and specification coverage. Note that `KeY-ABS` does not support several statements of `ABS`, such as **`case`**, `throw`, `try`,

[4] https://github.com/abstools/absexamples

Feature	Support	Specification	KeY-ABS	SN[1]
Asynchronous Contr.	Yes	`Requires`, `Ensures` on methods in interfaces	Yes[§]	No
Synchronous Contr.	Yes	`Requires`, `Ensures` on methods in classes	No	Yes
Cooperative Contr.	Yes	`Succeeds`, `Overlaps` on methods and **await** `Resolves` on **get**	No	Yes
Object Invariants	Yes	`Requires`, `ObjInv` on classes	Yes[†]	No
Function Contracts	Yes	`Requires`, `Ensures` on functions	No	No
Loop Invariants	Yes	`WhileInv` on loops	No	No
Session Types	Partial[a]	`Role` on classes, `Local` on methods	No	Yes
`old` and `last`	Yes		No	-
Counterexamples	Yes		No	-
History Specification	Partial[b]		Yes	-
First-order logic Specification	No		Yes	-
Exceptions and **assert**	Yes		Partial[‡]	-
Functions and ADTs	Yes		No	-

[a] No passive choice and exceptions. [b] Can be partially encoded by hand. [1] `SN` = static nodes.
[§] Encodable. [†] No creation condition. [‡] Only `null` access

Table 1: Overview over feature support in `Crowbar` for coreABS.

and **assert**.[5] For a detailed discussions on the limitations on the underlying ABSDL logic we refer to Kamburjan [31, Ch.2]. A further point worth mentioning is that `KeY-ABS` does not support specification within the program and takes it as additional input. In contrast, `Crowbar` supports all this and connects specification and program tightly by using annotations for specifications.

Limitations. `Crowbar` is limited to partial correctness of `coreABS`. Furthermore, `Crowbar` does not support passive choice and exceptions in local types, which are an open research question. Additionally, it does not support first-order specifications and specification of the history of events, which however can be added manually by the user: Histories can be added by introducing a special ADT for events and a field of list type in every class that managed explicitly. Due to its design, `Crowbar` relie on the counterexample generation to interact between user and proof, as it relies on an automatic SMT solver for its backend. Additionally, `Crowbar` can output the symbolic execution tree, as well as the concrete SMT-LIB output.

[5] A reason for the limitations is that it reuses `KeY`-Java internally, which do not support, for example, functional structures, and encode assumptions that are valid for Java, but not ABS directly in the code. An example for the last point is that in ABS, classes cannot be extended, while Java supports code inheritance.

```
class CeFrame {
 Int beats = 21239; String s = "o_3";
 Unit ce() {  // Snippet from: heartbeat
  String req = "fut_1";
  Int status = 5; //Int status = (req).get
  if((status == 200)){}
  else {// this.handleError();
  //Assume following effects while blocked:
    this.beats = 21238;
  }
  // Evaluates to: 21238
  println(toString(this.beats));
  // Failed postcondition:
  //      (heap.beats>=old.beats)
  //  /\ (result=heap.beats)
  // Failed to show the sub-obligations:
  // (select(anon(heap), this.beats)
  // >=heap.beats)
 }
}
```

Figure 5: An example for counterexample generation with `Crowbar`.

Related Tools. We already discussed `KeY-ABS` in detail above. `Chisel` [41] is a tool for *hybrid* Active Objects that generates proof obligations in $d\mathcal{L}$ [42], which in turn has three implementations as `KeY`-style symbolic execution engines [43] for a minimal hybrid programming language.

Outside the `KeY`-family, the Why3 [44] and Boogie [45] frameworks provide deductive verification middle-ends based on SE. Their separation of verification technique and programming language makes them not suited for our situation: we target a specific concurrency model with a *tight* coupling of specification and verification.

For Active Objects, Rebeca [46] supports model checking [47], which is limited to bounded systems and supports no modular specifications. Regarding deductive verification, `KeY-ABS` and `Crowbar` are the only systems for Active Objects. For actors, there exists a proposed program logic by Gordon [48] and a number of static analyses for Erlang, for which we refer to an overview in the work of Bagherzadeh et al. [49]. Note, however, that pure actors are not object-oriented, and that they do not implement cooperative scheduling or futures.

5. Illustrative Example

We give an example how to specify, verify and investigate a program and continue with the code specified in Lst. 1. The following attempts verification:

```
> crowbar example.abs --method Module.Monitor.heartbeat -inv
```

The switch `--method` verifies a single method and requires its fully qualified name, in this case `Module.Monitor.heartbeat`. The proof attempt fails: There is no contract given for `handleError`, so it is not specified how the method changes the field `heartbeat`. The `-inv` flag activates the counterexample generator, which outputs the counterexample shown in Fig. 5. All context interactions of the method (calls, suspension, synchronization, etc.) are removed to ensure that it can be executed on its own. Furthermore, it adds comments about extracted values and what interactions have been removed. Here, it shows that if the value of `heartbeat` was 21239 before and `handleError` changes it to 21238, then a part of the post-condition does not hold. The code is executable: as all context is removed, the programer may now examine and manipulate the counterexample using standard debugging techniques.

6. Conclusion

Impact. `Crowbar` is an important step in the empirical research on analysis of Active Objects: it is the first verification tool to cover full `coreABS` and implements novel specification and verification techniques in a flexible framework that allows one to investigate further new approaches with little overhead. In particular, we are now able to verify feature-rich programs, such as the C extraction case study. For potential impact, `Crowbar` enables us to tackle the numerous ABS case studies focusing on their specification without modifying them to fit the very restricted fragment supported by prior tools.

Future Work. We plan to use Crowbar to implement novel heavyweight symbolic execution systems, in particular the probabilistic dynamic logic of Pardo et al. [50], for which development has started, and a concurrent setting for the dynamic logic for memory access patterns [51]. As for planned extensions, we plan to integrate delta-oriented verification [52] next, as well as integrate first-order specifications and automate the handling of histories.

Bibliography

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, M. Ulbrich (Eds.), Deductive Software Verification - The KeY Book - From Theory to Practice, Vol. 10001 of LNCS, Springer, 2016. `doi:10.1007/978-3-319-49812-6`.

[2] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, R. Hähnle, Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case, in: CAV (1), Vol. 9206 of LNCS, Springer, 2015, pp. 273–289.

[3] S. de Gouw, F. S. de Boer, R. Bubel, R. Hähnle, J. Rot, D. Steinhöfel, Verifying OpenJDK's sort method for generic collections, J. Autom. Reason. 62 (1) (2019) 93–126.

[4] F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, A. M. Yang, A survey of active object languages, ACM Comput. Surv. 50 (5) (2017) 76:1–76:39. `doi:10.1145/3122848`.

[5] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: A core language for abstract behavioral specification, in: FMCO, Vol. 6957 of LNCS, Springer, 2010, pp. 142–164. `doi:10.1007/978-3-642-25271-6\_8`.

[6] G. Turin, A. Borgarelli, S. Donetti, E. B. Johnsen, S. L. T. Tarifa, F. Damiani, A formal model of the kubernetes container framework, in: T. Margaria, B. Steffen (Eds.), ISoLA, Vol. 12476 of Lecture Notes in Computer Science, Springer, 2020, pp. 558–577. `doi:10.1007/978-3-030-61362-4\_32`.

[7] J. Lin, I. C. Yu, E. B. Johnsen, M. Lee, ABS-YARN: A formal framework for modeling hadoop YARN clusters, in: P. Stevens, A. Wasowski (Eds.), FASE, Vol. 9633 of Lecture Notes in Computer Science, Springer, 2016, pp. 49–65. `doi:10.1007/978-3-662-49665-7\_4`.

[8] J. Lin, M. Lee, I. C. Yu, E. B. Johnsen, Modeling and simulation of spark streaming, in: L. Barolli, M. Takizawa, T. Enokido, M. R. Ogiela, L. Ogiela, N. Javaid (Eds.), AINA, IEEE Computer Society, 2018, pp. 407–413. `doi:10.1109/AINA.2018.00068`.

[9] E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, P. Y. H. Wong, Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS, Serv. Oriented Comput. Appl. 8 (4) (2014) 323–339. `doi:10.1007/s11761-013-0148-0`.

[10] E. Kamburjan, R. Hähnle, S. Schön, Formal modeling and analysis of railway operations with active objects, Sci. Comput. Program. 166 (2018) 167–193. `doi:10.1016/j.scico.2018.07.001`.

[11] N. Bezirgiannis, F. S. de Boer, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, Implementing SOS with active objects: A case study of a multicore memory system, in: R. Hähnle, W. M. P. van der Aalst (Eds.), FASE, Vol. 11424 of Lecture Notes in Computer Science, Springer, 2019, pp. 332–350. `doi:10.1007/978-3-030-16722-6\_20`.

[12] C. Consortium, Compugene, www.compugene.tu-darmstadt.de (2019).

[13] C. C. Din, R. Bubel, R. Hähnle, KeY-ABS: A deductive verification tool for the concurrent modelling language ABS, in: CADE'25, Vol. 9195 of LNCS, 2015, pp. 517–526. `doi:10.1007/978-3-319-21401-6\_35`.

[14] C. C. Din, S. L. T. Tarifa, R. Hähnle, E. B. Johnsen, History-based specification and verification of scalable concurrent and distributed systems, in: ICFEM, Vol. 9407 of LNCS, Springer, 2015, pp. 217–233.

[15] E. Kamburjan, T. Chen, Stateful behavioral types for active objects, in: IFM, Vol. 11023 of LNCS, Springer, 2018, pp. 214–235.

[16] E. Kamburjan, C. C. Din, T. Chen, Session-based compositional analysis for actor-based languages using futures, in: ICFEM, Vol. 10009 of Lecture Notes in Computer Science, 2016, pp. 296–312.

[17] E. Kamburjan, C. C. Din, R. Hähnle, E. B. Johnsen, Behavioral contracts for cooperative scheduling (2020).

[18] E. Kamburjan, Behavioral program logic, in: TABLEAUX, Vol. 11714 of LNCS, Springer, 2019, pp. 391–408.

[19] R. Bubel, C. C. Din, R. Hähnle, K. Nakata, A dynamic logic with traces and coinduction, in: TABLEAUX, Vol. 9323 of LNCS, Springer, 2015, pp. 307–322.

[20] E. Kamburjan, Detecting deadlocks in formal system models with condition synchronization, in: AVOCS, Vol. 76 of ECEASST, 2018.

[21] E. Kamburjan, R. Hähnle, Deductive verification of railway operations, in: RSSRail, Vol. 10598 of LNCS, Springer, 2017, pp. 131–147.

[22] E. Kamburjan, N. Wasser, The right kind of non-determinism: Using concurrency to verify C programs with underspecified semantics 365 (2022) 1–16.

[23] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.

[24] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, R. Schlatte, Variability modelling in the ABS language, in: B. K. Aichernig, F. S. de Boer, M. M. Bonsangue (Eds.), FMCO, Vol. 6957 of Lecture Notes in Computer Science, Springer, 2010, pp. 204–224. doi:10.1007/978-3-642-25271-6_11.

[25] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, User-defined schedulers for real-time concurrent objects, Innov. Syst. Softw. Eng. 9 (1) (2013) 29–43. doi:10.1007/s11334-012-0184-5.

[26] E. Kamburjan, S. Mitsch, M. Kettenbach, R. Hähnle, Modeling and verifying cyber-physical systems with hybrid active objects, CoRR abs/1906.05704. arXiv:1906.05704.

[27] R. Schlatte, E. B. Johnsen, E. Kamburjan, S. L. T. Tarifa, Modeling and analyzing resource-sensitive actors: A tutorial introduction, in: F. Damiani, O. Dardha (Eds.), COORDINATION, Vol. 12717 of Lecture Notes in Computer Science, Springer, 2021, pp. 3–19. doi:10.1007/978-3-030-78142-2_1.

[28] R. Schlatte, E. B. Johnsen, J. Mauro, S. L. T. Tarifa, I. C. Yu, Release the beasts: When formal methods meet real world data, in: F. S. de Boer, M. M. Bonsangue, J. Rutten (Eds.), It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab, Vol. 10865 of Lecture Notes in Computer Science, Springer, 2018, pp. 107–121. doi:10.1007/978-3-319-90089-6_8.

[29] E. Kuiter, A. Knüppel, T. Bordis, T. Runge, I. Schaefer, Verification strategies for feature-oriented software product lines, in: P. Arcaini, X. Devroey, A. Fantechi (Eds.), VaMoS, ACM, 2022, pp. 12:1–12:9. `doi:10.1145/3510466.3511272`.

[30] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, W. Dietl, JML Reference Manual, draft revision 2344 (May 2013).

[31] E. Kamburjan, Modular verification of a modular specification: Behavioral types as program logics, Ph.D. thesis, Technische Universität Darmstadt (2020).

[32] C. C. Din, O. Owe, Compositional reasoning about active objects with shared futures, Formal Aspects Comput. 27 (3) (2015) 551–572.

[33] B. Beckert, D. Bruns, Dynamic logic with trace semantics, in: CADE, Vol. 7898 of LNCS, Springer, 2013, pp. 315–329.

[34] B. Beckert, A dynamic logic for the formal verification of java card programs, in: Java Card Workshop, Vol. 2041 of LNCS, Springer, 2000, pp. 6–24.

[35] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, N. Yoshida, Behavioral types in programming languages, Found. Trends Program. Lang. 3 (2-3) (2016) 95–230.

[36] C. C. Din, O. Owe, A sound and complete reasoning system for asynchronous communication with shared futures, J. Log. Algebraic Methods Program. 83 (5-6) (2014) 360–383.

[37] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: POPL, ACM, 2008, pp. 273–284.

[38] N. Rollshausen, Counterexample generation for formal verification of abs, Bachelor Thesis, TU Darmstadt, available at `https://tuprints.ulb.tu-darmstadt.de/17856/` (2021).

[39] `CLOC` tool, version 1.82.

[40] N. Wasser, A. H. Tabar, R. Hähnle, Automated model extraction: From non-deterministic C code to active objects, Sci. Comput. Program. 204 (2021) 102597.

[41] E. Kamburjan, From post-conditions to post-region invariants: Deductive verification of hybrid objects, in: HSCC'21, ACM, 2021.

[42] A. Platzer, The complete proof theory of hybrid systems, in: LICS, IEEE Computer Society, 2012, pp. 541–550.

[43] S. Mitsch, A. Platzer, A retrospective on developing hybrid system provers in the KeYmaera family - A tale of three provers, in: 20 Years of KeY, Vol. 12345 of Lecture Notes in Computer Science, Springer, 2020, pp. 21–64.

[44] J.-C. Filliâtre, A. Paskevich, Why3 — where programs meet provers, in: M. Felleisen, P. Gardner (Eds.), ESOP, Vol. 7792 of Lecture Notes in Computer Science, Springer, 2013, pp. 125–128.

[45] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: FMCO, Vol. 4111 of LNCS, 2005.

[46] M. Sirjani, A. Movaghar, A. Shali, F. S. de Boer, Modeling and verification of reactive systems using rebeca, Fundam. Informaticae 63 (4) (2004) 385–410.

[47] M. Sirjani, A. Movaghar, A. Shali, F. S. de Boer, Model checking, automated abstraction, and compositional verification of rebeca models, J. Univers. Comput. Sci. 11 (6) (2005) 1054–1082. doi:10.3217/jucs-011-06-1054.

[48] C. S. Gordon, Modal assertions for actor correctness, in: F. Bergenti, E. Castegren, J. D. Koster, J. Franco (Eds.), AGERE!@SPLASH, ACM, 2019, pp. 11–20. doi:10.1145/3358499.3361221.

[49] M. Bagherzadeh, N. Fireman, A. Shawesh, R. Khatchadourian, Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences, Proc. ACM Program. Lang. 4 (OOPSLA) (2020) 214:1–214:32. doi:10.1145/3428282.

[50] R. Pardo, E. B. Johnsen, I. Schaefer, A. Wasowski, A specification logic for programs in the probabilistic guarded command language, in: ICTAC, Vol. 13572 of Lecture Notes in Computer Science, Springer, 2022, pp. 369–387.

[51] R. Bubel, R. Hähnle, A. H. Tabar, A program logic for dependence analysis, in: IFM, Vol. 11918 of Lecture Notes in Computer Science, Springer, 2019, pp. 83–100.

[52] M. Scaletta, R. Hähnle, D. Steinhöfel, R. Bubel, Delta-based verification of software product families, in: E. Tilevich, C. D. Roover (Eds.), GPCE, ACM, 2021, pp. 69–82.

Required Metadata

Current code version

Ancillary data table required for subversion of the codebase. Kindly replace examples in right column with the correct information about your current code, and leave the left column as it is.

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1.1.2
C2	Permanent link to code/repository used for this code version	https://github.com/Edkamb/crowbar-tool/releases/tag/v1.1.2
C3	Permanent link to Reproducible Capsule	doi.org/10.24433/CO.6726262.v1
C4	Legal Code License	BSD-3-Clause
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Kotlin, gradle, antlr, github
C7	Compilation requirements, operating environments & dependencies	Z3, Java $\geq$ 1.11
C8	If available Link to developer documentation/manual	https://github.com/Edkamb/crowbar-tool/wiki
C9	Support email for questions	eduard@ifi.uio.no

Table 2: Code metadata (mandatory)