Separation Logic and Compositional Symbolic Execution: Verification and Bug Detection at Scale

Philippa Gardner

Lecture 3 - *June 26, 2026*

These are the notes for the third lecture of the OPLSS 2026 course on Separation Logic and Compositional Symbolic Execution, taught by Philippa Gardner. The official lecture slides are available at `https://gillianplatform.github.io/docs/oplss26/Lecture3.pdf`.

1 Compositional Symbolic Execution on WiSL

The linear memory model of the WHILE language is a useful demonstrator memory model for introducing separation logic. However, it has some important caveats, such as generating too many constraints about pointer arithmetic and buffer separation.

One way in which analysis tools for real-world languages reason about memory models tractably is by viewing the heap not as a map indexed by naturals, but rather as a collection of memory blocks, and treating pointers as block-offset pairs. Thus, we introduce WiSL, a more general language that is still very close to WHILE and satisfies this memory model.

$$\hat{\sigma} = (\hat{s}, \hat{\mu}, \widehat{\mathsf{pred}}, \hat{\pi})$$

where $\hat{s} : \mathsf{PVar} \rightharpoonup_{\text{fin}} \mathsf{LExp}$ is the symbolic store, $\hat{\mu} : \mathsf{LExp} \rightharpoonup_{\text{fin}} \mathsf{LExp}$ is the symbolic linear heap, $\widehat{\mathsf{pred}}$ stores user-defined predicate instances, and $\hat{\pi}$ is the symbolic path condition.

For Gillian to reason about memory models such as linear models (with or without unique branch matching, cut branching, or negative information), fractional ownership models, OOP models, and many more, we need to establish soundness conditions that relate concrete and symbolic models.

Compiled By:
Dinghong Zhong, Samuel Dodson, Daniel Andres Pinto Alvarado

The Over-Approximation (OX) Soundness Condition

OX Soundness enforces that all states reachable by concrete execution, are also reachable by symbolic execution (which allows for verifying software):

$$\sigma, C \Downarrow \sigma' \wedge \sigma \models \hat{\sigma} \implies \exists \hat{\sigma}', \hat{\sigma}, C \Downarrow \hat{\sigma}' \wedge \sigma' \models \hat{\sigma}'$$

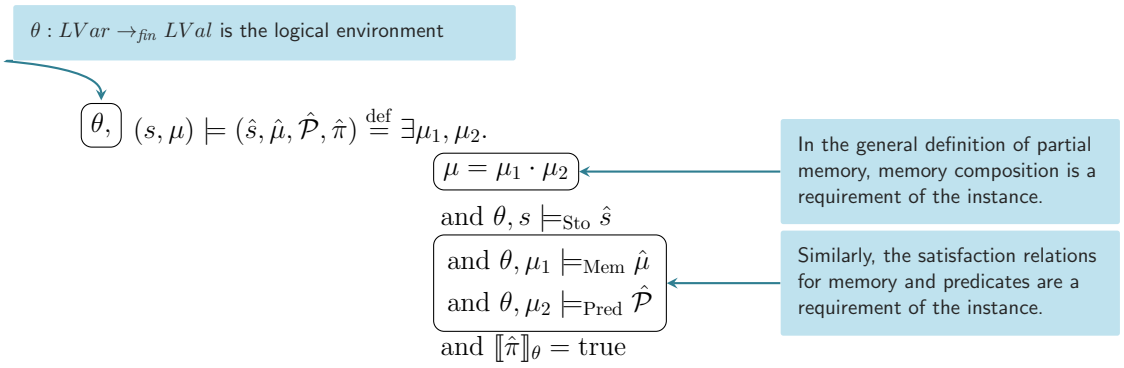
The Under-Approximation (UX) Soundness Condition

UX Soundness enforces the opposite condition, and thus it's particularly addequate for bug finding (i.e: ensure that a bug will arise in the concrete model provided that it shows in the symbolic one):

$$\hat{\sigma}, C \Downarrow \hat{\sigma}' \wedge \sigma' \models \hat{\sigma}' \implies \exists \sigma', \sigma, C \Downarrow \sigma' \wedge \sigma \models \hat{\sigma}$$

Generalizing memory soundness

Gillian extends the notion of soundness for its 4-tuple state:



Notice that on top of the soundness condition over the heap, we also require a soundness condition over the store and assertions. Moreover, we require another heap for the assertion since they might predicate over memory: $list(x, \alpha)$.

2 Extending Proof Rules to WISL

To formally reason beyond proof sketches, we also need to extend the proof rules. For that, we use the notation $\mu.\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o' : (\mu', \vec{v}')$ for an action α, which is run over the memory μ, uses the

parameters $\vec{v}$, and results in an outcome o (which can be ok, $error$, $miss$...), a memory μ', and return arguments $\vec{v}'$.

OX frame *"If memory is removed by an action, (non-missing) behaviour is unaffected"*

If $(\mu \cdot \mu_f).\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o : (\mu', \vec{v}')$

then $\exists\, \mu'', \vec{v}'', o'.\ \mu.\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o' : (\mu'', \vec{v}'')$ and

$(o' \neq miss \Rightarrow (o' = o \text{ and } \vec{v}'' = \vec{v}' \text{ and } \mu' = \mu'' \cdot \mu_f))$

UX frame *"If memory is added by an action, (non-missing) behaviour is unaffected"*

If $\mu.\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o : (\mu', \vec{v}')$ and $o \neq miss$ and $\mu' \cdot \mu_f$ is defined

then $(\mu \cdot \mu_f).\alpha(\vec{v}) \rightsquigarrow_{\mathcal{M}} o : (\mu' \cdot \mu_f, \vec{v}')$

Although the UX frame rule is fairly straightforward, the OX frame rule is more subtle. For UX, we can add more memory μ_f without affecting behaviour as long as the action did not result in a $miss$: if we can exhibit a bug using memory μ, then we can also exhibit it using more than μ. For OX, to extend analysis results to larger states, we say that removing a frame μ_f from $\mu \cdot \mu_f$ results in either a $miss$ outcome or the same behaviour as executing from the full state.

Moreover, we also need to "lift" the soundness condition into our new context (which is more easily expressed as making the following diagrams commute):

OX soundness

All *concrete* actions can be simulated by a corresponding *symbolic* action

If

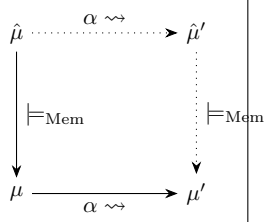
$$\mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}')$$

and

$$\theta, \mu \models_{\text{Mem}} \hat{\mu}, \qquad [\![\vec{E}]\!]_\theta = \vec{v}$$

then

$$\exists \hat{\mu}', \hat{\pi}', \vec{E}', \theta'.\ \hat{\mu}.\alpha(\vec{E}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{E}').$$



UX soundness

All *symbolic* actions can be simulated by a corresponding *concrete* action

If

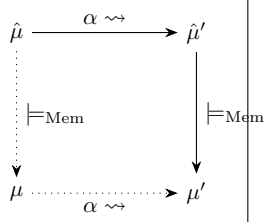
$$\hat{\mu}.\alpha(\vec{E}) \rightsquigarrow o : (\hat{\mu}', \hat{\pi}', \vec{E}')$$

then

$$\exists \mu, \theta.\ \theta, \mu \models_{\text{Mem}} \hat{\mu}$$

and

$$\mu.\alpha(\vec{v}) \rightsquigarrow o : (\mu', \vec{v}').$$



3 CSE Using Function Specifications

Inlining every function call is not a scalable basis for symbolic execution. Recursive functions may not terminate under naive inlining, library functions may not have available source code, and even ordinary helper functions can explode the number of symbolic paths. Function specifications address this by summarizing a call in the vocabulary of separation logic: the precondition says which resources the caller must provide, and the postcondition says which resources the caller receives back.

This style of reasoning is especially important for pointer-based data structures. A caller should be able to use a list, tree, object, or buffer operation through an abstract predicate, rather than by replaying the internal heap manipulations of the callee. The symbolic executor therefore needs a sound way to connect paper-style specifications with executable operations on symbolic states.

3.1 Function Specifications

A function specification has a precondition and one or more labelled postconditions. A typical specification has the following form:

$$\Gamma(f)|_m = \{\vec{x} = \vec{x} \star P\}\ f(\vec{x})\ \{\mathsf{ok} : Q_{\mathsf{ok}}\}\ \{\mathsf{err} : Q_{\mathsf{err}}\}.$$

Equivalently, in a syntax closer to Gillian specifications:

```
[ P ] f(x) [ ok: Qok ] [ err: Qerr ]
```

Here P describes the resources required to call f, Q_{ok} describes the resources produced by a successful call, and Q_{err} describes the resources produced by an error outcome. The assertion P can use ordinary separation logic predicates, such as list predicates, tree predicates, or predicates for a language-specific object model.

3.2 Symbolic Rule for Function Calls

For a call

```
y := f(E1, ..., En)
```

the symbolic semantics proceeds by consulting the specification rather than by executing the body of f. For the normal branch, the rule has the following shape.

1. Look up the function specification $\Gamma(f)|_m$ for the current mode m.
2. Symbolically evaluate the actual parameters:

   $$[\![\vec{E}]\!]_{\hat{s}} \Downarrow \hat{\vec{E}}, \qquad \hat{\theta} = [\vec{x} \mapsto \hat{\vec{E}}].$$

   Add the constraint $\hat{\vec{E}} \in \mathsf{Val}$ to the path condition, so the call only proceeds on symbolic states where the actual arguments denote runtime values.
3. Consume the precondition P from the current symbolic state:

   $$\mathsf{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\mathsf{ok}, \hat{\theta}', \hat{\sigma}_f).$$

   Intuitively, this checks that the current symbolic state can provide the resources required by P. If it can, those resources are separated from the rest of the state, leaving the untouched frame $\hat{\sigma}_f$. This is a proof-search step over symbolic resources, not a runtime deletion of heap cells.

4. Introduce a fresh logical return value $\hat{r}$, substitute it for ret in the normal postcondition, and extend the logical environment with the return binding.

5. Produce the normal postcondition in the frame:

$$\mathsf{produce}(Q_{\mathsf{ok}}[\hat{r}/\mathsf{ret}], \hat{\theta}'[\mathsf{ret} \mapsto \hat{r}], \hat{\sigma}_f) \rightsquigarrow \hat{\sigma}'.$$

6. Update the caller's symbolic store with $y \mapsto \hat{r}$.

The error branch is analogous, except that it produces Q_{err} instead of Q_{ok}. The important point is that the caller never needs the callee's body at the call site: it needs only the specification and the consume/produce operations.

3.3 Consume and Produce Interface

The algebraic interface records exactly the facts needed to prove soundness of constructs defined using consume and produce, including function calls, fold/unfold commands, and lemmas. The interface has four properties:

OX consume	If a concrete state satisfies the original symbolic state and symbolic consume does not force an abort on satisfiable paths, then there is a successful symbolic consume that splits the concrete memory into a consumed part μ_P satisfying P and a frame μ_f satisfying the leftover symbolic state.
UX consume	If symbolic consume succeeds, and concrete memories separately satisfy the consumed assertion and the produced frame, then their composition satisfies the original symbolic state.
OX produce	If a concrete resource satisfies the assertion being produced and can be composed with a concrete frame satisfying the old symbolic state, then symbolic produce can construct a new symbolic state satisfied by the composed concrete memory.
UX produce	If symbolic produce returns a symbolic state satisfied by some concrete memory, then that concrete memory can be split into a part satisfying the produced assertion and a frame satisfying the original symbolic state.

These properties are independent of the source language, the concrete memory model, and the implementation of the symbolic executor. They say what consume and produce must mean algebraically, rather than how a particular tool must implement them.

3.4 General Soundness Definition

The general OX/UX soundness definitions are independent of any particular construct. OX soundness says that all concrete executions represented by an initial symbolic state can be simulated by a corresponding symbolic execution:

$$\sigma, C \Downarrow \sigma' \wedge \sigma \models \hat{\sigma} \quad \Rightarrow \quad \exists \hat{\sigma}'.\ \hat{\sigma}, C \Downarrow \hat{\sigma}' \wedge \sigma' \models \hat{\sigma}'.$$

UX soundness gives the converse simulation direction. If a symbolic execution can reach a symbolic final state with a concrete instance, then there is a concrete initial state represented by the symbolic initial state whose execution reaches that concrete final state:

$$\hat{\sigma}, C \Downarrow \hat{\sigma}' \wedge \sigma' \models \hat{\sigma}' \quad \Rightarrow \quad \exists \sigma.\ \sigma, C \Downarrow \sigma' \wedge \sigma \models \hat{\sigma}.$$

3.5 Status Across Memory Models

Compositional symbolic execution does not combine every language's heap representation, ownership discipline, and primitive memory actions into one universal memory model. Instead, it separates the generic CSE proof obligations from the memory-model instance. Each instance supplies its own notion of memory shape, memory composition, satisfaction, and actions such as lookup, mutation, allocation, and deallocation.

Linear memories, fractional ownership, C block-offset memories, object-oriented models such as JavaScript objects, CHERI-style memories, and block-of-trees models for C are therefore different instantiations of the same interface, not parts of one combined heap model. Each instantiation must be checked for the relevant OX/UX properties, and only some of them currently have Gillian implementations.