CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity (Artifact)

This document describes the artifact submitted in support of the PLDI submission “CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity”.

Getting Started

The artifact is a VirtualBox VM image that contains our Isabelle/HOL formalization, our implementation HyperViper, and the benchmarks from our evaluation.
To run it, simply import it into an up-to-date version of VirtualBox (we tested with version 6.1) that has the VirtualBox extension pack installed. It uses 8GB of RAM and four logical cores by default; if these values are too high for your system, feel free to adjust the number of cores, but the VM may not work correctly with less than 8GB of RAM.

The image contains an installation of Ubuntu 22.04. Both the user name and the password are “artifact”.

For a quick check to ensure that the setup works, we suggest that you try the following two steps:

Ensure Isabelle works

To get started, we recommend making sure that all the files are successfully verified by Isabelle.

Our mechanization is located in ~/commcsl, and it contains the following 10 Isabelle files:

- AbstractCommutativity.thy
- CommCSL.thy
- FractionalHeap.thy
- Guards.thy
- Lang.thy
- PartialMap.thy
- PosRat.thy
- Safety.thy
- Soundness.thy
- StateModel.thy

a. Using Isabelle’s CLI

One can check that Isabelle successfully verifies all 10 files using the Isabelle command line interface (located at *~/Isabelle2022/bin/isabelle*, accessible via the shortcut *isabelle*) with the command *"isabelle build -c -d. -l CommCSL"* (this command tells Isabelle to build the *CommCSL* session, which is defined in the *ROOT* file).

This can be achieved with the following command:

```
> cd ~/commcsl
> isabelle build -c -d. -l CommCSL
```

Expected output:

The final lines of the output should look like the following:

```
…
Session Unsorted/CommCSL
  /home/artifact/commcsl/AbstractCommutativity.thy
  /home/artifact/commcsl/CommCSL.thy
  /home/artifact/commcsl/FractionalHeap.thy
  /home/artifact/commcsl/Guards.thy
  /home/artifact/commcsl/Lang.thy
  /home/artifact/commcsl/PartialMap.thy
  /home/artifact/commcsl/PosRat.thy
  /home/artifact/commcsl/Safety.thy
  /home/artifact/commcsl/Soundness.thy
  /home/artifact/commcsl/StateModel.thy
Cleaned CommCSL
Running CommCSL …
Finished CommCSL (0:01:29 elapsed time, 0:05:06 cpu time, factor 3.43)
0:01:35 elapsed time, 0:05:06 cpu time, factor 3.21
```

This output indicates that Isabelle successfully verified the 10 files in 1 minute and 35 seconds (it might take a bit longer). A different output might indicate a problem.

b. Using Isabelle's GUI

Note that Isabelle's GUI, which is located at *~/Isabelle2022/Isabelle2022,* can also be used to ensure that Isabelle can verify all files. To verify that a file is successfully verified:

1. Open the file (File > Open…).
2. Open the Theories panel (Plugins > Isabelle > Theories panel). It should be visible on the right of the window.
3. Activate "continuous checking" by ticking the box at the top of the Theories panel.
4. Put the cursor at the end of the file.

The verification status can be seen on the right of the editor, next to the scrollbar:

- Pink indicates a part that has not been verified yet.
- Purple indicates ongoing verification.
- Clear or orange indicates successful verification. Orange indicates a warning (warnings do not indicate invalid proofs, but correct proofs that can be optimized).
- Red indicates an error (this should *not* happen).

Ensure HyperViper works

HyperViper is installed in the VM and can be used by running the file *hyperviper.sh* located in ~/hyperviper.
To make sure it runs correctly, run it on one of the examples located in ~/hyperviper/evaluation. For example, run

```
> cd ~/hyperviper
> ./hyperviper.sh evaluation/01_count-vaccinated.vpr
```

Expected output:

The output should look like this:

Silicon 1.1-SNAPSHOT (<revision>@<branch>)

Silicon finished verification successfully in 13.91s.

This output indicates that HyperViper has verified the given file successfully in 13.91 seconds. Note that, as we will explain below, HyperViper is implemented as a plugin for Viper, so the executable being run is one of Viper's backends (called Silicon); this is the reason for the lack of the name HyperViper in the output.

Artifact overview

In this document, we will first give an overview of the data available in the artifact.
Then we will describe the details of our Isabelle formalization and how it maps to the formalization in the paper.
Subsequently, we describe the general usage and specification language (as well as some implementation details) of HyperViper, and explain how to replicate our evaluation.

The desktop of the VM contains a copy of the submitted paper.
The home folder of the artifact user in the VM contains the following important files and directories:

- The ~/commcsl directory contains our Isabelle formalization.
- The ~/hyperviper directory contains the source code of our tool HyperViper. HyperViper has been pre-installed and can be run by simply executing hyperviper.sh path/to/file.vpr on the command line from the ~/hyperviper directory (see below).
- The ~/hyperviper/evaluation directory contains the examples from our evaluation (Table 1); each example corresponds to the same file with the respective name. Most of these files use some shared data structure implementations and specifications imported from the ~/hyperviper/library directory.

Our artifact fully supports the following claims made in the paper:

- The artifact contains the entire Isabelle mechanization of CommCSL and its soundness proofs. That is, it supports our claim about a mechanized logic and mechanized correctness proofs from Sections 3 and 4, and it contains proofs for all theorems and lemmas in the paper.
- The artifact contains the implementation of HyperViper, the prototype verifier that automates CommCSL as described in Section 5.
- The artifact contains the examples we verified in our evaluation in Table 1 along with all specifications and supports our claims about verified examples, used data structures and abstractions, as well as all claims made about particular examples.

Our artifact also partially supports the performance claims we make in the paper in Table 1; naturally, a VM with limited resources will lead to slower verification times, but the artifact does substantiate the general trend of the verification times and gives upper bounds.

Step by Step Instructions

In this section, we will (1) explain the details of the Isabelle mechanization of CommCSL, and (2) explain how to use HyperViper in general and how to replicate the evaluation in Section 5 of the paper.

Structure of the Isabelle mechanization

Language

- Lang.thy: defines the syntax and semantics of the concurrent programming language presented in the paper. This file is adapted from Viktor Vafeiadis' CSL proof of soundness (https://people.mpi-sws.org/~viktor/cslsound/), which was ported to Isabelle 2016-1 by Qin Yu and James Brotherston.

State model and assertion language

- PosRat.thy: defines the type of positive rationals used as permissions.

- PartialMap.thy: defines and proves properties of partial maps (used to define permission heaps).
- FractionalHeap.thy: defines and proves properties of permission heaps.
- StateModel.thy: defines the guard heaps and the addition of guard heaps described in the paper.
- CommCSL.thy: defines the assertion language of CommCSL.
- Guards.thy: defines and proves properties about the guard states.

Soundness

- Safety.thy: defines the safe predicate described in the paper, and proves some results about it.
- AbstractCommutativity.thy: proves lemma C.2 from the paper (Appendix C).
- Soundness.thy: proves the soundness of all CommCSL rules.

Additionally, the file NonInterference.thy contains a non-interference theorem added for the final version of the paper, but was not contained in the original submission.

Correspondence with the paper

We suggest to use Isabelle's GUI to navigate the mechanization (see how in the *Getting Started Guide*), in order to check that it is consistent with the claims in the paper. To jump to the definition of a term, click on it while holding the Control key.

Note the following differences between the formal descriptions in the paper and the Isabelle/HOL mechanization:

Invariants (*)

In the paper, we write I(v) to state that the invariant I holds and maps its state to some value v. In Isabelle, these two aspects are separate: a function from partial heaps to values (called "view" throughout) defines v, and there is an additional assertion View that states what the current value of the view is, and there are constraints that the view must be defined for partial heaps that satisfy the invariant.

Assertions (**)

The Isabelle formalization contains an assertion NoGuard P, which enforces that P is fulfilled only by states without guards; this is used to ensure that the noguard(P) constraints in proof rules that are also shown in the paper an be easily fulfilled.
PREa(s) is defined in terms of other assertions in the paper, but not in Isabelle.

The table below connects the claims in the paper with the Isabelle/HOL mechanization. To show the line numbers, click on "View > Toggle Line Numbers".

Paper		Isabelle/HOL mechanization	
Section	**Element(s)**	**File**	**Element(s)**
3.1	Figure 6 (programming language)	Lang.thy	datatypes exp, bexp, cmd (lines 13-35)
	Program state		type state (line 11)
	Semantics (App. A.1)		inductive definitions red, red_rtrans, aborts
3.2	Definition 3.1 (validity of resource specification)	AbstractCommutativity.thy	definition all_axioms (lines 9-23) (***)
3.3	Extended heaps	StateModel.thy	type heap (line 15)
	Addition of extended heaps		function plus (lines 135-138)
	Normalization of heaps	FractionalHeap.thy	definition normalize (lines 98-99)
3.4	Assertion syntax	CommCSL.thy	datatype assertion (lines 26-49) (**)
	Validity of assertions (figure 7)		function hyper_sat (lines 54-86) (**)
	Definition 3.2 (PRE_s)		lines 69-71 (**)
	Definition 3.2 (PRE_i)		lines 72-73 (**)
	unary		definition unary (lines 477-478)
3.5	Resource context	Guards.thy	record single_context (lines 15-20) (*)
	consistency, criterias (1) and (3)		definition semi_consistent (lines 50-51)
	consistency, criteria (2)	Safety.thy	definition sat_inv (lines 14-15)
3.6	Rules (figure 8)	Soundness.thy	inductive definition syntactic_hoare_triple (lines 4557-4596)
4	Definition of safe	Safety.thy	function safe (lines 214-226)
	Definition 4.1 (Hoare triples)		definition hoare_triple_valid (lines 565-567)
	Theorem 4.2 (Soundness)	Soundness.thy	theorem soundness (lines 4599-4601)
	Corollary 4.3		corollary safety_no_frame (lines 4508-4521)

App. A	Operational semantics (figure 9)	Lang.thy	inductive definitions red and red_rtrans (lines 77-97)
App. B	Sum of to unique guard heaps	StateModel.thy	definition add_gu (lines 46-47)
	Sum of shared guard heaps		function add_gs (lines 57-61)
	Permission heap addition	FractionalHeap.thy	definition add_fh (lines 83-84)
	Proof rules (figure 10)	Soundness.thy	inductive definition syntactic_hoare_triple (lines 4557-4596)
App. C	Definition of safe	Safety.thy	function safe (lines 214-226)
	Lemma C.2	AbstractCommutativity.thy	theorem main_result (lines 1469-1476)

(***) Because unique and shared actions have a different type in Isabelle, part (A) from the paper corresponds to 2 lines in Isabelle, and part (B) to 3 lines. On top of what is presented in the paper, we also require that the preconditions are symmetric in their arguments.

Usage and Evaluation of HyperViper

General usage

As described in the Getting Started section, HyperViper can be used by invoking
./hyperviper.sh path/to/file/vpr
from inside directory ~/hyperviper.
HyperViper does not take any additional command line parameters, and will report either successful verification or an error message; examples of this are shown in the following section.

Replicating the evaluation

To replicate the the evaluation, in particular, the data in Table 1 of the paper, execute
> cd ~/hyperviper
> ./benchmark.sh

The expected result of running this command is contained in ~/hyperviper/benchmarkoutput.txt. This command will first verify two warmup files (warmup1.vpr and warmup2.vpr), which can be ignored, to convince the JVM to JIT-compile HyperViper's code, and subsequently verify all examples in the ~/hyperviper/evaluation folder. These files are numbered, so the examples are verified in the same order they are shown in Table 1.

The output generated by benchmark.sh will (after some initial setup output) look as follows for each example:

[info] - /home/artifact/hyperviper/commutativity-plugin/src/test/resources/commutativity-tests/evaluation/**01_count-vaccinated.vpr** [Silicon Statistics]
[info] + [Benchmark] Time requires: 51 msec (Parsing), 32 msec (Semantic Analysis), 17 msec (Translation), 272 msec (Consistency Check), 9.30 sec (Verification), **9.68 sec (Overall)**

The relevant parts are the name of the example and the overall verification time.
Note that all examples are intended to verify correctly, with the exception of 04_figure-1.vpr, which produces a verification error (since it is insecure and would leak information). When running benchmark.sh, this file will nevertheless be shown in green, like all other examples, since it contains an annotation that states that a verification failure is expected.
The expected output ends with a statement that all tests passed.

To verify any individual example, execute e.g.
./hyperviper.sh evaluation/04_figure-1.vpr
as described above.

For all examples except example 04_figure-1.vpr, the output should be

Silicon 1.1-SNAPSHOT (<revision>@<branch>)

Silicon finished verification successfully in 13.91s.

*(*modulo different timing).
For example 04_figure-1.vpr, the expected output is

Silicon 1.1-SNAPSHOT (<revision>@<branch>)

Silicon found 1 error in 4.26s:
[0] The precondition of method print might not hold. Assertion p1_2 && p2_2 ⇒ tmp1_1 == tmp2_1 might not hold. (04_figure-1.vpr@45.3)

which states that verification fails because a precondition of a method call in line 45 cannot be established.
Note that, as HyperViper is currently a prototype, error messages do not always indicate the problem in a readable way; we are working on better error back-translation for HyperViper.
Also note that verification times will be longer when running HyperViper on a single example than when verifying an example as part of a longer run using benchmark.sh, since individual runs include JVM startup time and do not leave the JVM enough time to warm up and JIT-compile the code. As stated in the paper, the verification times listed in Table 1 are

measured on a warmed-up JVM, i.e., they are the times we get when verifying several examples in a row.

IMPORTANT: We have found that on the virtual machine, unlike on any native environment we have tested HyperViper on, verification of some examples can (very rarely) non-deterministically fail. Failing examples will be marked red when running benchmark.sh, and may either show an unexpected verification error or a stack trace. Whether or not this happens seems to depend on the physical computer running the virtual machine. We have not been able to identify the cause of this (it is likely that the used SMT solver runs into some internal timeout). If this happens on your machine, rerunning either the individual test or the test suite will almost certainly resolve the issue.

To confirm the other information in Table 1, one has to inspect the example files manually. In particular, each file contains at least one resource specification (the syntax of which we will describe below) which defines the data structure, the valid actions, and the abstraction used by the example (which are listed in the table). The count of lines of code and annotations was performed manually. Crucially, the lines shown in Table 1 *include* the code and annotations of any file imported by the example files. That is, if a file starts with
import "../library/array.vpr"
then the lines of code and lines of annotations shown in Table 1 include those in the file itself, plus the code/annotations in said file array.vpr (which defines common methods and functions related to arrays).

Other claims we make about our implementation and our examples are:

- That HyperViper supports a more complex language than CommCSL as described in the paper: We will explain this claim in the following section.
- That ca. half of our examples contain code that branches on high information. This is the case in
  1) all five examples that use a hashmap: a map lookup mapGet first selects a bucket based on the hash value of the given key, and then walks through all values in said bucket (see also a comment in 02_figure-2.vpr that explains this). Whether it continues walking or not always depends on whether the current key is the one that it is looking for, and this information is high in all five examples.
  2) high branches occur in the two-producer-two-consumer example, where each consumer branches on the current size of the queue, which is high, and
  3) in all four examples that use lists, since the implementation of list append again walks through the current list (which, during concurrent modification, is high) and appends at the end; however, how long the list is at this time, and thus how long it has to walk, is high.
- That we can reuse the same resource specification for two different data structure implementations: Examples 09_sick-employee-names.vpr and 10_website-visitors use a resource specification that is completely identical except for the used invariant (which determines the data structure implementation). Note that in HyperViper, the resource

specification is declared alongside an invariant, unlike in the paper, where the resource specification is separate. Thus, the resource specification as defined in the paper are completely identical in both examples.

- That different examples declare different *aspects* of the used data to be low: “the data contains some low parts (e.g., whether or not a patient has been vaccinated), or some aspect of the secret data is low (e.g., the range of an employee’s salary or the number of—otherwise secret—purchases they have made).”
  The vaccination data is low in example 01, the range of an employee’s salary (but not the salary itself) is low in example 13, and the number of purchases (but not the purchases themselves) is low in example 14.

Syntax and specifications supported by HyperViper

HyperViper is implemented as a plugin for the open source Viper verifier (https://www.pm.inf.ethz.ch/research/viper.html). That is, it takes Viper’s ordinary language (which is a simple sequential language with support for a mutable heap) and its specification language and extends them with custom language and specification constructs for commutativity-based information flow reasoning.
Note that the code example shown in App. E of the paper was slightly modified to help readability, and thus uses slightly different terms than HyperViper’s actual implementation in some cases (though the structure of the code in App. E is unchanged and correct).

Resource specifications

Resource specifications are top-level declarations in HyperViper. Since HyperViper supports multiple shared resources in a single program by supporting multiple locks (and each lock is associated with a resource specification), they are called lock types in HyperViper. A simple resource specification declaration looks as follows:

```
lockType CounterLock {
 type Int
 invariant(l, v) = [l.lockCounter |-> ?cp && [cp.val |-> v]]
 alpha(v): Int = v

 actions = [(Incr, Unit, duplicable)]

 action Incr(v, arg)
   requires true
 { v + 1 }

 noLabels = N()
}
```

Similar to the presentation in the paper, they declare the logical type of the value they work with (here that type is Int), an abstraction function alpha, and a set of actions (here only one, which is called Incr and takes Unit as its parameter type), each of which is a function marked as shared

(called duplicable in the tool) or unique. In HyperViper, one first declares the set of actions in a given lock type and marks them shared or unique, and subsequently defines the actions and their (potentially relational) preconditions.
Unlike in the paper, in HyperViper, lock types are immediately associated with invariants (which map the value of the resource to some concrete implementation, in this case, to the value of the field cp.val).
Additionally, lock type declarations contain an integer noLabels, which defines how many times the guards for the shared actions in the lock type should be splittable; this is used to aid automation (see also the respective ghost statements below). In particular, in HyperViper, each shared action guard is associated with a set of integer labels 0 … noLabels. Performing the action requires a guard with a singleton set containing a specific label, and having the entire guard corresponds to having the guard with the full set of labels. Guards have to be explicitly split and merged using ghost statements (see below).

Additionally, lock types can contain proof blocks that define proofs (in the form of intermediate assertions the prover should check that will enable it to prove the overall proof goal) to help HyperViper prove aspects of the validity of each declared lock type. In most cases, HyperViper is able to prove these properties completely automatically, but if the automatic check e.g. of action commutativity fails, users can manually write such a block to help the prover. An example from example 13_salary-histogram.vpr that helps HyperViper prove the commutativity of two disjoint put actions Put1 and Put2 looks as follows:

```
proof commutativity[Put1, Put2](v, arg1, arg2) {
 var r1 : MyMap[Int, Int] := put(put(v, fst(arg1), snd(arg1)), fst(arg2),
snd(arg2))
 var r2 : MyMap[Int, Int] := put(put(v, fst(arg2), snd(arg2)), fst(arg1),
snd(arg1))
 assert map_eq(r1, r2)
}
```

Statements

HyperViper extends the standard Viper language with several new statements:

- t := fork m(args)
  Forks a new thread that executes method m with arguments args, and assigns the resulting thread object to variable t. As we state in the paper, while CommCSL only supports parallel composition statements c || c, HyperViper supports dynamic forking and joining of threads.
- join[m](t)
  Joins thread object t, which must have executed method m.
- with[LT] l when e performing a(x) at lbl { c }
  Acquires lock l of lock type LT once expression e is true, executes statement c, and releases the lock again. All other parts of the statement are annotations that help the proof, i.e., they are required to fix values that are free in CommCSL rules, like the action a in the Atomic rules, s.t. HyperViper does not have to try to automatically find the

correct values. Here, a must be an action allowed by lock type LT, and statement c must change the state in a way that corresponds to executing action a with argument x (and all of these properties are, of course, checked by the tool). The end “at lbl” is only used if a is a shared action: In this case, it updates the specific action guard with integer label lbl.

- share[LT](l, e)
  Shares the previously not shared lock l of lock type LT with an initial value of e (which requires that the lock’s invariant holds, consumes it, and creates all action guards for the lock).
- unshare[LT](l)
  Unshares the previously shared lock l of lock type LT, which consumes the action guards and checks, among other things, that all action preconditions were fulfilled (i.e., PRE holds for every action), and lets the user assume that the abstraction of the data protected by the lock is low afterwards.
- merge[LT, a](l, s1, s2)
  Merges two guards for shared action a of lock l of type LT. One guard must have the set of labels s1, the other the set s2, and the merged guard will have the set s1 union s2. This statement is again required as an annotation from the user, to avoid that the prover has to automatically infer when to merge (and split) shared action guards.
- split[LT, a](l, s1, s2, ms1, ms2)
  Performs the opposite of merge and splits the currently held guard for shared action a of lock l of lock type LT (which must have the union of the labels in s1 and s2 and whose argument multiset must be the union of multisets ms1 and ms2) into two separate guards with label sets s1 and s2 and with action argument multisets ms1 and ms2.

Assertions and expressions

HyperViper adds several new assertions and expressions to the Viper language:

- A points-to assertion [e.f |-p-> ?v && A] states that receiver e has a field f which points to some existentially-quantified value v, and assertion A holds for said value. Additionally, the optional expression p (which defaults to 1) expresses the fractional permission amount to this field that the current context owns.
- The relational assertion low(e) states that the value of e is low.
- The relational assertion lowEvent states that the current calling context is low, i.e., whether or not the current point in the program is reached does not depend on high data. This assertion is not supported in CommCSL but is useful to specify effectful methods, e.g., for a print method, one may want to ensure not only that the printed value is low, but also that whether or not something is printed at all does not depend on high data. This property can be expressed using the method precondition lowEvent.
- The assertion sguard[LT, a](l, s) represents a partial guard for shared action a of lock l of type LT. s is a set of labels, i.e., if s is the set 0 … noLabels, then the guard is a full guard and not a partial one. While in the paper, we have a guard assertion sguard(p, ms) where p is a fractional permission amount and ms is the multiset of action arguments, here, we have to additionally specify the lock (since we can have more than one) and

the action and lock type (since we can have more than one). We express the permission amount via the set of labels. The argument multiset ms is not included in the assertion in HyperViper, but handled separately via an expression:

- The expression sguardArgs[LT, a](l, s) represents the multiset of arguments in the partial guard for shared action a has been performed on lock l of type LT with label set s. This expression is well-defined only when the respective guard is held (and HyperViper reports an error if it is used in a context when the guard is not held). Separating the guard itself from its argument expression simplifies automation.
- Similarly, there is an assertion uguard[LT,a](l) that represents a full guard for unique action a of lock l of type LT, and its argument sequence is represented by the expression uguardArgs[LT, a](l).
- allPre[LT, a](e) is equivalent to PREa(e) in the paper and expresses that the multiset or sequence of arguments e satisfies the precondition of action a.
- Finally, joinable[m](t, args) states that t is a thread running method m with arguments args and can be joined.

HyperViper implementation overview

As stated above, HyperViper is implemented in Scala as a plugin for the open source Viper verification infrastructure. It extends Viper's syntax with custom constructs for commutativity-based information flow reasoning. Users can write programs using said extended syntax, and HyperViper subsequently encodes it into a standard Viper program, which is then verified by Viper's default symbolic execution backend, which ultimately uses the SMT solver Z3.

Directory ~/hyperviper contains the implementation of HyperViper in multiple parts:

- ~/hyperviper/silver contains the definition of Viper's standard verification language, and ~/hyperviper/silicon contains Viper's standard execution backend. These are parts of the open source Viper infrastructure and used without major modifications.
- ~/hyperviper/commutativity-plugin contains the entire implementation of HyperViper itself, which will be described further below.
- ~/hyperviper/commutativity-plugin-test contains no code itself and exists only for build purposes; it has Viper itself and the HyperViper plugin as dependencies and thus can be packaged to get a single jar file.
- ~/hyperviper/silver-sif-extension is a pre-existing open source implementation of the modular product program transformation for Viper, and is used without major modifications by HyperViper. This transformation enables Viper, which is unable to directly verify relational properties, to verify programs containing relational Low(e)-assertions.

The main project, commutativity-plugin, contains six main files:

- CommutativityPluginASTExtensions.scala and CommutativityPluginPASTExtensions.scala contain definitions of AST nodes for the

additional statements, expressions and assertions introduced by HyperViper. For example, CommutativityPluginASTExtensions.scala contains a class LockSpec (representing a lock type, i.e., a resource specification) that contains a type, a definition of an abstraction function, and several action definitions and an invariant declaration.

- CommutativityErrors.scala contains classes representing custom error types relating to commutativity-based reasoning, e.g., an error type that represents a failed commutativity check.
- CommutativityParser.scala defines an extension of Viper's standard parser to parse the newly-added statements, declarations, expressions and assertions.
- CommutativityPlugin is the main class of HyperViper that defines its extension of the Viper language in the form of a plugin. It hooks into Viper's extension mechanism by
    - extending its parser (by overriding the beforeParse method and adding the new parse rules defined in CommutativityParser.scala)
    - desugaring special types for locks and threads to simple references (by overriding the beforeResolve method, which is called before type checking)
    - and encoding all added language constructs to the standard Viper language (by overriding method beforeConsistencyCheck, which is called after type checking and before verification, to call the main encoding method defined in CommutativityTransformer.scala).
- CommutativityTransformer.scala defines the encoding of the extended Viper language with constructs for commutativity-based information flow reasoning to the standard Viper language. Its main method, encodeProgram, performs three main steps:
    - It checks various consistency criteria on the input program by calling checkDeclarationConsistency (which ensures, for example, that resource invariants do not contain Low-assertions).
    - It generates definitions and declarations of various helper functions and predicates which are used in the encoding. In particular, this includes the definition of PRE_a for each action a, which is generated by method generateAllPre.
    - It encodes proof obligations relating to the well-definesness and validity of all declared lock specifications by calling method encodeExtension.
    - It defines the encoding of all added statements (e.g., for forking and joining threads and acquiring resources) and assertions (e.g. relating to guards) in the inner functions transformStmt and transformExp, separately for each statement and assertion. For example, lines 457-503 encode the proof obligations resulting from a share-statement into regular Viper code. The generated code asserts that the lock invariant holds, asserts that the supplied initial value of the shared data structure is low modulo abstraction, and creates guards for all actions of the newly-shared lock. For each encoded statement and assertion, comments in the code show the generated Viper code.
    - Finally, it applies this transformation to the input program in line 781, and subsequently, since the generated program still uses relational Low(e) and LowEvent assertions, calls outside code to construct a product program of the

generated program, which also transforms Low-assertions into regular Viper assertions.

Extending HyperViper

HyperViper can easily be extended to work with additional statements, assertions, or more complex resource specification by performing the following steps:

1. Add a new AST and ParseAST node for the new construct (or adapt the existing one) in CommutativityASTExtensions.scala and CommutativityPASTExtensions.scala. The ParseAST node internally has to define how to type check the new node, analogously to the existing nodes.
2. Define a new parse rule in CommutativityParser.scala, and add it to the newStmt, newExp, or newDecl definitions at the top of the file.
3. If the new construct requires any kind of desugaring before type checking, extend method beforeResolve in CommutativityPlugin.scala to do so accordingly.
4. In CommutativityTransformer,
    a. For new top-level declarations or extended lock specifications, extend method checkDeclarationConsistency to add all required additional syntactic checks.
    b. For new top-level declarations or extended lock specifications, extend method encodeExtension to generate all added proof obligations resulting from the new or extended declaration.
    c. Generate any additional helper definitions inside encodeProgram.
    d. Extend methods transformExp and/or transformStmt (by adding a case that matches on the new expression or statement type) inside encodeProgram to define the encoding of added statements, expressions or assertions to standard Viper code.