Copper Manual

Introduction

The general goal of Copper is to verify that an implementation conforms to its specification. The implementation is always a C program and hence quite standard. However, Copper can be used to play around with several kinds of specifcations and notions of conformance. For a quick overview of how to start using Copper, check out the tutorial.

Usage

Copper is a command line tool. It is invoked with a set of options and input file names. The input files are either pre-processed C files with extension .pp or specification files with extension .spec. The C files must be pre-processed with CIL. The specification files must obey this grammar. The most basic way to run Copper is as follows:

```
copper  --default --specification <specification name> <filenames>
```

The above invocation will cause Copper to check simulation conformance between the FSP specification identified by "specification name" and the program defined by the input C files. To verify SE-LTL specifications, use the following command line:

```
copper  --default --ltl --specification <specification name> <filenames>
```

Other common types of verification tasks are: (1) checking trace containment between a program and an FSP specification, and (2) checking for the unreachability of ERROR states in an FSP specification. These are achieved, respectively, by the following two command lines:

```
copper  --default --trace --specification <specification name> <filenames>
```

```
copper  --default --reach --specification <specification name> <filenames>
```

In reality, "--default" is just a synonym for the following set of options which, most often, appear to work best in practice:

```
--useAllSpecs --ceShowAll --stat --cegar --symbolic
```

The general usage of Copper is:

```
copper <option | filename> <option | filename> ...
```

In general, anything on the command line that begins with a minus sign is treated as an option. Everything else is treated either as an argument of the preceeding option or a filename. Later options take precedence over earlier ones whenever applicable. Thus, "--trace --reach" is equivalent to "--reach". Thus, "--default" should be one of the first options you supply to Copper. We now describe the set of all available Copper options in more detail. Options are grouped together into broad categories. Options within the same category

are more closely related to each other than options belonging to a different category.

Options

In the following we write "by default" to mean the situation when no command line options are supplied to Copper, and not the situtation when the option "--default" is provided. The following options affect the **global behavior** of Copper.

- **--help** or **-h**: Generate this help message and exit.
- **--specification** <name>: The name of the specification to be verified.
- **--copperHome** <copper home directory>: Home directory of the copper distribution. This value overwrites the COPPER environment variable, if set.
- **--parse**: Only parse input files and exit.
- **--echo**: Print characters as they are being parsed. This is helpful for debugging parse errors.
- **--stat**: Display statistics at the end. Implied by "--default".
- **--verbose** <level>: Set the verbosity level. Default is 2.
- **--timeLimit** <number>: Time limit in seconds. Default value is 10000 hours.

The following options control the **type of verification** to be carried out. By default, Copper checks simulation between the C program and the specification.

- **--reach**: Check ERROR state reachability instead of simulation.
- **--trace**: Check trace containment instead of simulation.
- **--ltl**: Do SE-LTL model checking.
- **--deadlock**: Do deadlock detection.
- **--assert**: Check iteratively for reachability of assertions of the form "assert(0)" in the program. In each iteration, Copper checks for the possible reachability of one such assertion.

In the rest of this document, we refer to trace containment, ERROR state reachability, and the detection of assertion failure collectively as "safety" verifications since all of them boil down to model checking safety properties. The following options control the **verification mechanism** used by Copper. By default, Copper uses a HORNSAT based model checker for simulation and safety verification, and an explicit state model checker for SE-LTL model checking and deadlock detection.

- **--explicit**: Use explicit state model checking instead of HORNSAT for verification.
- **--symbolic**: Use symbolic predicate abstraction to reduce the number of theorem prover calls. Implied by "--default". Use only for safety verification and SE-LTL model checking.
- **--bp**: Generate boolean programs via abstraction and use BDDs for verification. Use only for safety verification and SE-LTL model checking.

The following options control the type of **Assume-Guarantee** reasoning done by Copper. By default, Copper does not perform Assume-Guarantee reasoning. Also, Assume-Guarantee reasoning is only supported for simulation, safety verification, and deadlock detection.

- **--ag**: Do assume-guarantee reasoning with learning for verification.
- **--agc1a**: Use circular rule 1A for assume-guarantee style reasoning.
- **--noAGReuse**: Do not reuse counterexamples to candidate queries. This may increase the number of candidate queries.
- **--agOpt**: Assume-guarantee reasoning with minimal assumption alphabet.
- **--agLazy**: Assume-guarantee reasoning with lazy alphabet extension.

The following options control the degree of **inlining** used by Copper. By default, Copper inlines according to the "inline" directives in the specification files.

- **--inline**: Inline all library routines whose code is available.
- **--pds**: Use pushdown system models instead of inlining. Only use for safety verification and do not combine with "--bp".

The following options control the initial set of "seed" **predicates** used by Copper. Seed predicates are essentially branch conditions from which other predicates are inferred at various control flow points by propagating weakest preconditions. By default, there are no initial seed predicates and the only initial predicates are those appearing in the specification.

- **--autoPred**: Automatically add all branches in the code as seed predicates.
- **--specSeed**: Obtain initial seed predicates from spec files.

The following options control the abstraction **refinement** used by Copper. By default, Copper does no refinement and will terminate with the model checking result on the first abstract model.

- **--cegar**: Do abstraction refinement without predicate optimization. Implied by "--default".
- **--optPred**: Do pseudo-boolean constraint based predicate optimization.
- **--greedy**: Do greedy predicate optimization. In some cases, this leads to quicker termination compared to "--optPred".
- **--usefulPred**: Use only useful predicates for predicate discovery. A useful predicate is one appearing in a "predicate" directive in the specification files. This option enables the user to guide abstraction refinement by only allowing Copper to pick new "seed" predicates from those specified in the "predicate" directives. However, this mechanism is only useful if the user has some idea about the kinds of "seed" predicates on which the program's correctness is based.
- **--ceDag** <number>: Use this many counterexamples in each iteration for refinement. Default is 1. If the argument after "--ceDag" is "n" then Copper generates "n" counterexamples during model checking and checks each of them for validity. If even one counterexample is found to be valid, Copper terminates with that real "bug". Otherwise, if all the counterexamples are found to be spurious, Copper refines the abstraction to eliminate all of them. Larger values of "n" may lead to quicker termination, but also to increased resource consumption.
- **--predLoop** <number>: Number of times to go through a loop during predicate discovery. Default value is 1. Larger values will lead to more predicates being inferred from the same set of "seed" predicates, and hence a more precise model and more effective abstraction refinement. However, this may also lead to increased resource consumption.

The following options control the **theorem prover** used by Copper. The default theorem prover is Simplify.

- **--cprover**: Use CProver as the theorem prover. This is the only appropriate option if you want support for full bit-level semantics and all C operators.
- **--cvc**: Use CVC as the theorem prover. Only available on Linux.
- **--ics**: Use ICS as the theorem prover. Only available on Linux.
- **--svc**: Use SVC as the theorem prover. Only available on Linux.
- **--cvcl**: Use CVC Lite as the theorem prover. Only available on Linux.
- **--vampyre**: Use Vampyre as the theorem prover. This is experimental and only available on Linux.
- **--cogent**: Use Cogent as the theorem prover. This is experimental.
- **--TPCache**: Cache theorem prover queries and their results. This could increase memory requirement.
- **--TPCacheSize** <size>: The size of the theorem prover cache. The cache is cleared if this size is exceeded. Default is 1000.

The following options control the **SAT solvers** used by Copper. By default, Copper uses ZChaff.

- **--sato**: Use Sato as the SAT solver.
- **--grasp**: Use FGrasp as the SAT solver.
- **--chaff**: Use ZChaff as the SAT solver.

The following options control the information displayed as part of the **counterexamples**. By default, only the sequence of the control locations from the program that appear in the counterexample are displayed.

- **--ceShowAct**: Show actions when displaying counterexamples.
- **--ceShowCons**: Show propositional constraints when displaying counterexamples.

--ceShowAll: Show actions and propositional constraints when displaying counterexamples. Implied by "--default".

These options control various **statespace reduction** techniques used by Copper for efficient verification:

- **--silentTrans**: Don't eliminate silent transitions. By default, Copper eliminates silent transitions before model checking to make verification more efficient. A silent transition is defined to be one that does not change the value of any specification proposition, and is not labeled by any specification action. Eliminating silent transitions leads to a type of partial-order reduction by avoiding many possible thread interleavings during model checking.
- **--noParAssign**: Disable parallel assignments. By default, Copper transforms a sequence of simple assignments into a single semantically equivalent parallel assignment. A parallel assignment is essentially a set of assignments executed concurrently. For example Copper translates the following sequence of assignments "temp = x; x = y; y = temp" into the parallel assignment "x = y || y = x || temp = x". Be aware that parallel assignments may lead to unsound results when the specification has propositions, and should be disabled in such cases.
- **--simParAssign**: Use naive (but correct, even with pointers) strategy for parallelizing assignments.
- **--eager**: Complete abstraction before model checking. The default is to abstract lazily on-the-fly during model checking.

The following options achieve **miscellaneous** purposes.

- **--checkCE**: Check validity of counterexample if not doing abstraction refinement. This enables you to detect if the counterexample found by verifying the first abstract model is valid or spurious.
- **--useAllSpecs**: Inline all library routine specifications. Overrides the default rule that only library specifications that contain synchronization or global specification actions are inlined. Implied by "--default".
- **--invariant**: Compute invariants via static analysis. Invariants are used for subsequent abstraction and verification.
- **--pointers**: Handle aliasing due to pointers.
- **--noSyntactic**: No syntactic checks before calling theorem prover.
- **--assign**: Assignment actions are not replaced by epsilons.
- **--return**: Return actions are not replaced by epsilons.

The following options control the aspects of Copper related to **proof** generation.

- **--proof**: Generate proofs when the specification is found to be valid. Use only for safety verification and SE-LTL model checking.

The following options control the aspects of Copper related to **proof-carrying code**. For these options to work, a version of gcc that generates PowerPC binaries must be installed as "gcc-ppc". Also, the target C program must be called "pcc-test.c".

- **--pcc**: Generate and prove verification conditions for a specific procedure in "pcc-test.c", which must be annotated with invariants. An invariant is simply a side-effect free C expression and is specified using the distinguished procedures "__begin__" and "__inv__". Specifically, a control flow point "Loc" is annotated with an invariant "X" by inserting the code "__begin__(); __inv__(X);" just before "Loc". It is recommended that you have as many control flow points annotated with invariants as possible. In particular, every loop in the control flow graph of the relevant procedure must contain at least one control flow point annotated with an invariant. Otherwise, Copper will not terminate. In practice, Copper first compiles "pcc-test.c" into PowerPC assembly using "gcc-ppc" and then generates and proves the verification conditions on the assembly level.
- **--pccProc** <procedure_name>: Procedure to check with "--pcc". Default is "main". Use only in combination with "--pcc".
- **--noPccAsm**: Skip compiling "pcc-test.c" to assembly. Use an existing assembly program, which must be called "pcc-test.s" and must have been generated previously by Copper using "--pcc". Use only in combination with "--pcc".

Copper Tutorial

Copper is a software model checker. It enables you to verify properties against C source code. Copper is also fairly versatile about the kinds of claims it can check. On one hand, it can look for simple errors such as assertion violations . On the other hand, you can use Copper to check more complicated specifications expressed as finite state machines or linear temporal logic formulas. Copper also supports the verification of multi-threaded programs where the threads communicate with each other via shared variables, or handshakes, or both. Finally, while Copper uses explicit-state verification by default, it can be used to perform symbolic BDD-based verifiction by techniques described in the section on "Symbolic Verification".

However, Copper also has some limitations. In particular, Copper does not support non-integral basic data types such as floats and doubles. It treats all variables of such unsupported types as integers. Copper also treats pointers in an unsound manner. In practice, this means that the result of running Copper on a pointer-manipulating program cannot be trusted in general. However, Copper is still useful as a bug-finding tool on such programs. In addition, Copper treats integers as unbounded quantities by default. Allowing unbounded integers is, in general, unsound because integers are represented as bit-vectors in practice. This is usually not a big issue, but can be avoided by techniques described later in the section on "Bit-Vector Semantics". Finally, Copper is unable to handle recursive programs by default since it uses inlining. Techniques to verify recursive programs with Copper are described later in the section on "Recursion".

We recommend that you have Copper installed and setup to use this tutorial to its fullest potential. We use the well-known Dining Philosophers problem as a running example for learning how to use Copper. The Dining Philosophers problem consists of *n* philosophers and *n* forks placed alternately around a table. Thus, there is exaclty one fork between two successive philosphers and exactly one philosopher between two successive forks. We first model check philosophers and forks individually. Later on, we model check an assembly consisting of two philosophers and two forks.

Modeling a Philosopher

Each philosopher repeats the following steps ad-infinitum:

1. Picks up the fork to his right
2. Picks up the fork to his left
3. Eats
4. Puts down the fork to his left
5. Puts down the fork to his right
6. Goes back to Step 1

Then the following C procedure models a philosopher:

```
void philosopher()
{
   int eating;
   eating = 0;
   while(1) {
      pick_left();
      pick_right();
      eating = 1;
      if(eating != 1) assert(0);
      eating = 0;
      put_left();
```

```
    put_right();
  }
}
```

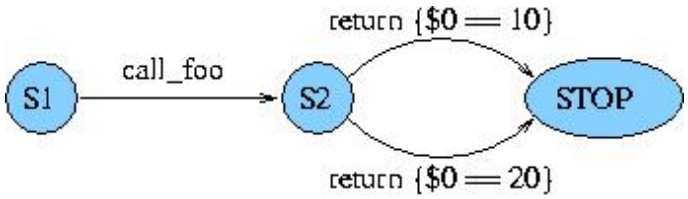
Note that we have used a variable **eating** to indicate whether the philosopher is currently eating or not. In addition, we have introduced an assertion for describing how Copper can be used to find assertion violations. Copper only supports assertions of the form **assert(0).** Thus, we assert any condition **cond** as **if(!cond) assert(0).**

Specifying Procedures

In practice, almost all programs we verify are incomplete. In other words, there are procedures (typically, library routines) called by the target program for which we don't have the source code. In the philospher example above, the procedures for acquiring and releasing forks are examples of such undefined procedures. Copper provides a mechanism for specifying the behavior of undefined procedures in terms of finite state machines, also known as Labeled Transition Systems (LTSs). During verification, these procedure specifications are used by Copper to construct appropriate models which are subsequently analyzed. We now describe the process of specifying LTSs, followed by the mechanism for associating these LTSs with procedures.

LTS Syntax

Copper uses an *FSP-like* notation to specify LTSs. It does not support the full FSP syntax. On the other hand, it supports certain features that are not available in FSP. More details can be found in the grammar for the LTS syntax. Below, we show an LTS, followed by its expression in Copper's notation:



S1 = (call_foo -> S2),
S2 = (return {$0 == 10} -> STOP | return {$0 == 20} -> STOP).

The name of a state machine is the name of its initial state. The transitions of the state machine are labelled with *actions.* We use actions to capture externally observable behavior, such as acquiring and releasing resources, sending messages, returning values, etc. Actions form the basis of synchronization between concurrently executing threads of a program. Actions, along with constraints on data variables, also serve as the building blocks of program specifications. The transition from the initial state (**S1**) is labelled by an action **call_foo**. Such actions, with only names, are also called *basic actions.*

IMPORTANT: The name of a state machine must always begin with a capital letter. This is followed by zero or more letters and digits. These and other details can be found in the specification grammar.

Return actions

We extend the FSP notation to express a special class of actions called *return actions.* Return actions are of the form **return {expression}** or **return {}** where the former expresses the return of an integer value and the latter expresses returning a *void* value. In a return action of the form **return {expression}**, the **expression** represents a condition satisfied by the return value. The return value itself is represented by the dummy variable **$0**. For instance, the action **return {$0 < 5}** represents the return of an integer value that is strictly

less than **5**.

Linking LTSs and Procedures

Returning to our philosopher example, we now wish to specify the behaviors of the undefined procedures. First, we wish to specify that the procedure **pick_left** performs the action **pick_left** and returns a **void** value. To do this, we first write down an LTS that encapsulates the desired behavior:

```
PickLeft = ( pick_left -> return {} -> STOP ).
```

Note that two or more actions in a row imply the existence of an unspecified state between every pair of consecutive actions. Thus, in the above, there is an unspecified state between the actions **pick_left** and **return{}**. Next, we associate the procedure **pick_left** with the LTS **PickLeft**:

```
procedure pick_left { abstract { 1, PickLeft }; }
```

Note the keywords **procedure** and **abstract**. The block keyword **procedure** indicates that we are going to say something about a C procedure. It is followed by the name of the procedure and then by a set of statements enclosed within a pair of curly braces. Each such statement typically consists of a *statement keyword* followed by other terms. The procedure whose name follows **procedure** is often referred to as the scope procedure. The **abstract** keyword indicates that we are expressing an abstraction relation between the scope procedure and an LTS. An abstract statement consists of two elements: (1) a guard which is a C expression constraining the calling context, and (2) the LTS to be associated. These two elements are enclosed by curly braces and separated by commas.

In the example above, the guard is **1** and the LTS is **PickLeft**. Note that **1** denotes *True* according to C semantics. This means that, according to the above abstraction, the LTS **PickLeft** specifies the behavior of the procedure **pick_left** under all calling contexts. In general, Copper allows dummy variable **$1**, **$2**, etc. to be used in guards, where **$i** refers to the *i-th* argument of the scope procedure. Thus, the following specification means that the behavior of procedure **foo** is modeled by the LTS **Foo1** when **foo** is callled with a non-zero argument and by the LTS **Foo2** otherwise.

```
procedure foo { abstract { ($1 != 0), Foo1 }; }
procedure foo { abstract { ($1 == 0), Foo2 }; }
```

Of course, we would also have to define the LTSs **Foo1** and **Foo2** separately. In addition, the procedure **foo** would always have to be invoked with at least one argument. Multiple procedure blocks can be combined into one as long as they have the same scope procedure. Also the order of statements within a procedure block is irrelevant. Thus, the above two procedure blocks together is equivalent to the following single procedure block:

```
procedure foo {
    abstract { ($1 != 0), Foo1 };
    abstract { ($1 == 0), Foo2 };
}
```

Copper requires that the guards of abstraction statements for any scope procedure be mutually disjoint and complete (i.e. cover all possibilities of argument valuations). This is necessary to enable Copper to unambiguously identify the applicable abstraction in any given calling context of the scope procedure. We now specify the behaviors of the remaining undefined procedures in our philosopher example:

```
PickRight = ( pick_right -> return {} -> STOP ).
procedure pick_right { abstract { 1, PickRight }; }

PutLeft = ( put_left -> return {} -> STOP ).
procedure put_left { abstract { 1, PutLeft }; }
```

```
PutRight = ( put_right -> return {} -> STOP ).
procedure put_right { abstract { 1, PutRight }; }
```

IMPORTANT: So far, we have associated every procedure with an LTS that simply performs an action and returns a void value. In practice, this is so common that Copper provides a shortcut without having to write any specifications. Specifically, suppose we want to associate a procedure **foo** with an LTS that peforms **bar** and returns. We can do this by simply replacing every call to **foo** in the program with a call to a special procedure **__COPPER_HANDSHAKE__** with the argument **"bar".** Thus, in our philosopher example, we can replace the call to **pick_left()** with **__COPPER_HANDSHAKE__("pick_left")** and so on for the other procedure calls. Remember this is just a shortcut. You can always use LTSs, or specify some procedures using LTSs and others using **__COPPER_HANDSHAKE__,** as we have done in philosopher.pp and philosopher.spec.

Specifying Programs

It is now time to specify the entire program that we want to verify. In our case the program is sequential, i.e. it has a single thread consisting of the procedure **philosopher**. We first describe how to write down claims (or properties) that we want to verify. We next show how to associate claims with programs. The nature of a claim depends on the kind of property we wish to check. Currently, Copper supports the following types of verification:

1. **Simulation** conformance between a program and an LTS specification.
2. **Trace containment** between a program and an LTS specification.
3. A program contains no trace that would cause an LTS specification to **reach an ERROR state**.
4. A program satisfies a **linear temporal logic** specification.
5. A program does not **violate an assertion**.
6. A program does not **deadlock**.

LTS Claims

In the case of checking simulation, trace containment or ERROR state reachability, the claim is simply an LTS. For instance, suppose we want to ensure that the philosopher always performs the actions **pick_left** and **put_left** alternately, starting with **pick_left**. Then we can check that it is simulated by the following specification LTS:

```
PhilSpec1 = ( pick_left -> put_left -> PhilSpec1 ).
```

IMPORTANT: Simulation can be viewed as *tree containment,* and hence is stronger than trace containment when the specification is non-deterministic. For deterministic specifications, simulation and trace containment are equivalent. Thus, for the above specification, we can check trace containment instead of simulation.

Note that the LTS **PhilSpec1** only refers to the actions **pick_left** and **put_left** and is blind to all other actions. Thus, the trace **T1 = (pick_left , pick_right , put_left)** does not violate **PhilSpec1.** But suppose that we also want to ensure that the philosopher does not perform a **pick_right** between a **pick_left** and **put_left**. Clearly, the specification **PhilSpec1** is incorrect. However, we can obtain a correct specification by simply *extending the alphabet* of the specification with **pick_right** as follows:

```
PhilSpec2 = ( pick_left -> put_left -> PhilSpec2 ) + { pick_right }.
```

In general, you can extend the alphabet of any state machine by adding a plus sign and a list of actions enclosed within curly braces at the end of the description of the initial state of that state machine. Now, the specification no longer ignores **pick_right** and therefore is violated by the trace **T1** defined above. Indeed, while our philosopher satisfies **PhilSpec1** it fails **PhilSpec2**.

ERROR State Reachability

Observe that the trace **T1** is a specific violation of the more general specification **PhilSpec2**. In practice, we are often interested in the presence of such specific "buggy" behaviors in programs. Copper provides a direct mechanism for such bug-hunting via ERROR state reachability. In this mechanism, we write down an LTS specification, such that any "buggy" trace takes you from the initial state to a state whose name begins with "ERROR". For instance, suppose that in the case of the philosopher, two successive occurences of **pick_left** without an intermediate **put_left** is undesirable. We can specify this with the following LTS:

```
PhilSpec3 = ( pick_left -> pick_left -> ERROR3 ) + { put_left }.
```

We then check if our program contains the undesirable behavior via ERROR state reachability using the above specification. In essence, when doing ERROR state reachability, Copper checks if the program contains a trace that corresponds to some path in the specification from the initial state to an ERROR state. Such traces, by definition, exhibit program bugs.

LTL Claims

Copper supports temporal logic claims expressed in State/Event Linear Temporal Logic (SE-LTL). The syntax of SE-LTL is similar to that of LTL, except that the atomic formulas are either actions or expressions involving program variables. In addition, SE-LTL supports the standard logical operators of conjunction (&), disjunction (|) and negation (!), as well as the next-time (#X), until (#U), globally (#G), eventually (#F) and release (#R) temporal operators. The syntax of SE-LTL follows that of LTL. An expression atomic formula is treated exactly as a proposition. Thus, it satisfies an infinite trace if and only if it holds on the first state of the trace. In contrast, an action atomic formula satisfies an infinite trace if and only if it is the first action to occur on the trace. The semantics of the logical and temporal operators is exactly the same as in the case of LTL. For example, suppose we want to check that whenever the philosopher is eating, it eventually always releases its left fork. This is specified by the following Copper claim:

```
ltl PhilSpec4 { #G ( [P0::eating == 1] =>  #F put_left ); }
```

Similarly, the claim that every occurrence of **pick_left** is eventually followed by another occurence of **pick_left** is expressed by the following claim:

```
ltl PhilSpec5 { #G ( pick_left =>  ( #X ( #F pick_left ) ) ); }
```

Note that the **#X** operator is important in the above claim to ensure that every **pick_left** is eventually followed by a *different* occurrence of **pick_left.** If we had omitted the **#X** operators, the claim would always be trivially satisfied. Also, when writing down SE-LTL claims, data constraints are always enclosed within square braces, but actions are not. Further details about the SE-LTL can be found in the specification grammar. Note that in **PhilSpec4**, the occurrence of "eating" is preceeded by "P0::". Intuitively, this means that this claim refers to the variable "eating" of the first (and this case, only) thread of our program. The significance of this notation will become clearer in the next section, where we define the process of associating claims with programs.

IMPORTANT: When checking for assertion violations and possible deadlocks, the specifications are implicitly defined and hence do not need to be specified.

Connecting Programs and Claims

The following *program block* associates each of the Philosopher claims defined above with the Philosopher program.

```
program philosopher {
    specification abs_1, {1}, PhilSpec1;
    specification abs_2, {1}, PhilSpec2;
    specification abs_3, {1}, PhilSpec3;
    specification abs_4, {1}, PhilSpec4;
    specification abs_5, {1}, PhilSpec5;
```

}

This looks a lot like a procedure block but there are some crucial differences. First, it begins with the keyword *program* instead of *procedure.* This is followed by a list of *procedure names.* Intuitively these are the names of the procedures which execute in parallel and constitute the program. In the above block this list has a single procedure name viz. **philosopher***,* signifying that our program has just one thread that executes **philosopher.** Following the list of procedure names we have a sequence of specifications enclosed in curly braces. A specification begins with the *specification* keyword, and has three other elements. The first is the name of the specification. This is used by Copper to identify the target specification, and hence the target program, to be validated. The second is a *list of guards,* one for each thread of the program. Each guard in the list expresses the beginning state of the corresponding thread. In the above block, the list has just one element that expresses the starting context of **philosopher.** Note that the list of guards is enclosed within curly braces. The third and final element is the name of the LTS which specifies the program.

IMPORTANT: Local and global variables must be referred to in specification files by prefixing them with an appropriate process identifier. This is important for disambiguation because Copper can verify concurrent programs and the same variable may appear in multiple threads. The general variable format is **Pn::v** where **n** is the thread number and **v** is the variables name. The threads are numbered in the order in which they appear in the **program** declaration, starting with zero. For example consider the following **program** declaration:

program foo,foo,bar { ... }

Then we refer to variable **v** of the first thread (**foo**) as **P0::v**, to variable **v** of the second thread (**foo**) as **P1::v** and to variable **v** of the third thread (**bar**) as **P2::v***.*

IMPORTANT: The *list of guards* element of a specification is particularly important when checking claims that involve program variables. During verification, all variables are assumed to have non-deterministic values in the initial state. This causes problems for simple claims like "x is always >= 0," for which Copper will report a counter-example in which the claim does not hold in the initial state. In such cases, the list of guards should be used to restrict Copper to exploration of cases in which the guards are satisfied. For example, by supplying **{P0::x == 0}** as a list of guards for a specification, Copper will only consider cases in which x *initially* has a value of 0.

Comments

You can use either C-style or C++ style comments in specification files.

/* this is a comment */
// so is this one

Running Copper

We are now ready to run Copper. First save the C program (which must be preprocessed using CIL) in a file whose name must end with ".pp", say philosopher.pp. Next save the specifications in another file whose name ends with ".spec", for example philosopher.spec. Finally run Copper. To check simulation between the program and **PhilSpec1** use the following command line:

$ copper --default --specification abs_1 philosopher.pp philosopher.spec

Copper tries to validate the specification with name **abs_1** which involves the claim **PhilSpec1.** The **--default** option instructs Copper to use the default set of options. For details on other options that Copper can accept, look at the user's manual. If all goes well, Copper should be able to successfully verify the specification and produce an output (here's how you should read Copper's output) that ends with something like this:

conformance relation exists !!
specification abs_1 is valid ...

To check trace containment with **PhilSpec1**, we use the additional option **--trace** as follows:

$ copper --default --specification abs_1 philosopher.pp philosopher.spec --trace

Once again, Copper proves trace containment and indicates this by the following output:

conformance relation exists !!
specification abs_1 is valid ...

Now, we move on to the second claim **PhilSpec2,** which we expect to fail. Indeed, running Copper with:

$ copper --default --specification abs_2 philosopher.pp philosopher.spec --trace

yields an output that contains a counterexample and the following indication of failure:

conformance relation does not exist !!
specification abs_2 is invalid ...

The next claim **PhilSpec3** is one of ERROR state reachability, and can be checked with the following:

$ copper --default --specification abs_3 philosopher.pp philosopher.spec --reach

Note that we use the option **--reach** instead of **--trace** and as expected Copper succeeds in proving that the claim holds on the program. Finally, we check the remaining two SE-LTL claims using the **--ltl** option.

$ copper --default --specification abs_4 philosopher.pp philosopher.spec --ltl
$ copper --default --specification abs_5 philosopher.pp philosopher.spec --ltl

As expected, both claims are found to hold. Finally, we can show that the assertion in **philosopher** can never be violated using the **--assert** option. Note that even though the specification for assertion violations is defined implicitly, we still have to mention an specification name to identify the program in which we wish to look for assertion violations. We achieve this via a "dummy" claim which, in our specification, looks like:

program philosopher { specification abs_6, {1}, DefaultSpec; }

We then supply the name of the specification (**abs_6** in this case) to Copper as follows:

$ copper --default --specification abs_6 philosopher.pp philosopher.spec --assert

We note that **DefaultSpec** is a special LTS pre-defined by Copper and hence we recommend its use for dummy claims. Similarly, dummy claims are also used for deadlock detection, as presented in the next section.

IMPORTANT: Assertion violations provide a mechanism for checking the reachability of a certain program location. Simply insert an **assert(0)** at that location and then check for assertion violations.

Reading Copper's Output

The execution of Copper is an iterative process since Copper attempts to verify increasingly refined models of its target program. If Copper terminates successfully, near the very end of the output you should find a line that says "**conformance relation exists !!**" or "**conformance relation does not exist !!**". The first alternative indicates that Copper found the claim being checked to hold on the target program. The second case indicates

that the target claim does not hold on the target program. In the second situation, the output also contains a counterexample just above the "**conformance relation does not exist !!**". Since Copper executes iteratively, it may print out several counterexamples while it runs. The last counterexample displayed by Copper is the one that actually shows the failure of the target claim. This counterexample starts with:

<<< BEGIN CONCRETE COUNTEREXAMPLE >>>

and ends with:

<<< END CONCRETE COUNTEREXAMPLE >>>

The body of the counterexample consists of an alternating sequence of states and actions, beginning and ending with a state. Actions are simply sandwiched between two lines that consist of a sequence of plus signs. For instance, the action **pick_left** is displayed as follows:

```
++++++++++++++++++++++++++++++++++++++++++++++++++
      pick_left
++++++++++++++++++++++++++++++++++++++++++++++++++
```

In contrast, a state is more complicated and consists of three distinct elements. The first element is sequence of control locations, one for each thread. The location for any thread corresponds to the statement that thread is *about to* execute. The second element is a set of data constraints indicating values of various propositions appearing in the specification. The second element is separated from the first by a line consisting of the pound or hash sign. The third element is an assignment to various program variables indicating their current values. The third element is separated from the second by a line consisting of the equals sign. For instance, consider the following state that appears in the output showing a deadlock in a system consisting of two philosopher threads and two fork threads (this example is described in more detail in the next section):

```
[<<-1>> P0::temp_var_10 = __COPPER_HANDSHAKE__ ( "pick_left_1"  ) <<[]>>:0:0]
[<<-1>> P1::temp_var_16 = __COPPER_HANDSHAKE__ ( "pick_right_2"  ) <<[]>>:65535:255]
[<<-1>> P2::temp_var_8 = do_fork1 (  ) <<[]>>:0:2]
[<<-1>> P3::temp_var_9 = do_fork2 (  ) <<[]>>:1:2]
################################################
================================================
(P1::eating = 0)(P0::eating = 0)
```

The first four lines show the statements that the four program threads (in the order they appear after the **program** keyword) are about to execute. The next line separates the first element from the second. After that, we immediately have the separator line between the second and third elements of the state. This is because the second element of the state is empty since, for deadlock detection, the specification has no propositions. Finally, we have the third element of the state which shows that the variables **eating** of the first two threads each have the value zero.

Modelling Philosophers and Forks

We now verify an assembly consisting of two philosophers and two forks. The key questions is: how the four threads communicate with each other. Copper supports a very general communication mechanism where threads can exchange data via global variables (as in threads) and also synchronize with each other by handshaking on a common action (as in CSP processes). Handshaking means that whenever a thread performs an action, say **foo**, it must do so jointly with all other threads that also have **foo** in their alphabets. If any of these other threads is unable to perform **foo** (from whatever state that thread is currently in), then **foo** cannot occur. We use the following convention when modeling our assembly:

- The philosopher procedures are called **phil1** and **phil2,** while the fork procedures are called **fork1** and **fork2**.
- **Phil1** has **fork1** to his right and **fork2** to his left. Hence, **phil2** has **fork2** to his right and **fork1** to his left.

- We model the picking and putting down of forks by actions of the form **pick_x_y** and **put_x_y** where **x** could be **left** or **right** and **y** could be **1** or **2**. Thus, the action **pick_right_2** models **phil2** picking up his right fork, which is **fork2**.

With this convention, the following procedures model the philosophers and the forks:

```
void phil1()
{
   int eating;
   eating = 0;
   while(1) {

__COPPER_HANDSHAKE__("pick_left_1");

__COPPER_HANDSHAKE__("pick_right_1");
      eating = 1;
      if(eating != 1) assert(0);
      eating = 0;

__COPPER_HANDSHAKE__("put_left_1");

__COPPER_HANDSHAKE__("put_right_1");
   }
}
```

```
void phil2()
{
   int eating;
   eating = 0;
   while(1) {

__COPPER_HANDSHAKE__("pick_left_2");

__COPPER_HANDSHAKE__("pick_right_2");
      eating = 1;
      if(eating != 1) assert(0);
      eating = 0;

__COPPER_HANDSHAKE__("put_left_2");

__COPPER_HANDSHAKE__("put_right_2");
   }
}
```

```
/* fork 1 */
void
fork1()
{

do_fork1();
}


/* fork 2 */
void
fork2()
{

do_fork2();
}
```

Of course we also need the following spepcification to define the procedures called by **fork1** and **fork2:**

```
DoFork1 = ( pick_right_1 -> put_right_1 -> DoFork1 | pick_left_2 -> put_left_2 -> DoFork1 ).
procedure do_fork1 { abstract { 1 , DoFork1 }; }
DoFork2 = ( pick_right_2 -> put_right_2 -> DoFork2 | pick_left_1 -> put_left_1 -> DoFork2 ).
procedure do_fork2 { abstract { 1 , DoFork2 }; }
```

Note that the forks are shared resources and hence can be possessed by at most one philosopher at any time. This is reflected by the definitions of **do_fork1** and **do_fork2** above. Look at files dp-2.pp and dp-2.spec for the complete program and specification. The claim we are interested to check is that both philosophers can never be eating at the same time. It is expressed as follows:

```
ltl DpSpec1 { #G [ (P0::eating == 0) || (P1::eating == 0) ]; }
```

We now run Copper:

```
$ copper --default --specification abs_1 dp-2.pp dp-2.spec --ltl
```

As expected, Copper succeeds in proving the claim. Note that we had to restrict the initial state of the program since local variables can have any value at the beginning and the claim would fail immediately if we do not restrict the initial values of **P0::eating** and **P1::eating**. We can now check for possible deadlocks in our assembly as follows:

```
$ copper --default --specification abs_2 dp-2.pp dp-2.spec --deadlock
```

As expected, Copper succeeds in finding a deadlock and even provides a counterexample that shows how a deadlock can occur. As in the case of assertion violations, we use a dummy claim for finding deadlocks since the actual claim is implicitly defined but we still have to specify the threads of the program and their initial states.

Miscellaneous Topics

We now describe some important issues related to Copper that we referred to earlier.

Symbolic Verification

As mentioned before, Copper uses explicit-state verification by default. It can be made to perform symbolic BDD-based model checking via the command line option "**--bp**". In case you are curious, "**bp**" stands for "Boolean Program", which is what Copper uses as an intermediate representation during symbolic verification. Since later options override earlier ones, be sure that "**--bp**" occurs after "**--default**" in your command line.

Bit-Vector Semantics

Copper uses the Simplify theorem prover by default, which treats integers as unbounded quantities. If you want bit-vector semantics use the command line option "**--cprover**". This forces Copper to use the Cprover theorem prover, which treats integers as 32-bit wide vectors, instead of Simplify. Be warned, however, that using Cprover typically makes Copper run much more slowly.

Recursion

By default, Copper inlines everything into one procedure, and hence is uanble to handle recursion. Use the "**--pds**" option to verify recursive programs. Again, "**pds**" stands for "Push-Down System" which is what Copper uses as an intermediate representation while verifying recursive programs. Copper only supports claims that have finite trace counterexamples (i.e., safety claims) when verifying recursive programs. In particular, this includes trace containment, ERROR state reachability and assertion violations, but excludes simulation, non-safety SE-LTL claims, and deadlock detection. Copper cannot verify recursive programs symbolically so do not combine "**--pds**" with "**--bp**".

In addition to *abstract,* there are several other keywords that can be used in procedure blocks for performing specific tasks. We now mention a few important ones.

Supplying predicates

The user can manually supply predicates to guide Copper's predicate abstraction. Often this is useful when Copper fails to discover a satisfactory set of predicates in a reasonable amount of time. Predicates are supplied on a per-procedure basis. In this regard, an important restriction is that all user-supplied predicates for a procedure **foo** must be syntactically equivalent to some branch condition in **foo**. Otherwise that predicate is simply ignored by Copper. For example consider the following C procedure:

```
int foo()
{
   int x = 5;
   if(x < 10) return -1;
   else return 0;
}
```

Suppose we want to prove using Copper that **foo** is correctly specified by the following LTS:

```
FOO = ( return {$0 == -1} -> STOP ).
```

Normally we would do this by simply asking Copper to perform automated abstraction refinement (using the **--optPred** option). However suppose we have a good idea about the predicates necessary for Copper to complete successfully. For example, in this case **(x < 10)** is the required predicate (note that this corresponds to a branch condition in **foo**). Then we can simply tell Copper to use this predicate by using the **predicate** keyword. The following procedure block shows how to do this:

```
procedure foo { predicate (x < 10); }
```

Copper looks for branch statements in **foo** which have the branch condition **(x < 10)**. If it finds any such branch, it uses the corresponding branch condition as a *seed predicate.* Otherwise it ignores the user supplied predicate. Multiple predicates can be supplied in one statement using a comma-separated list or they can be supplied via multiple predicate statements. Also, the order in which predicates are supplied is irrelevant. For example the two following procedure blocks each have the same effect as the procedure block above:

```
procedure foo {
    predicate (y == 10) , (w == 5) , (z +w > 20) , (x < 10) , (x+y != 5);
}

procedure foo {
    predicate (x+y != 5);
    predicate (z+w > 20) , (y == 10);
    predicate (x < 10) , (w == 5);
}
```

Inlining Procedures

Suppose procedure **foo** calls procedure **bar**. Normally Copper **does not inline bar** within **foo** even if the code for **bar** is available. It has to be told explicitly to do this via the **inline** keyword. Here's a procedure block that demonstrates how to do this. Once again inlining has to be done on a procedure-to-procedure basis. For example the following procedure block does not cause **bar** to be inlined within some other procedure **baz**.

```
procedure foo { inline bar; }
```

Questions and Comments

At this point, you should be more or less familiar with Copper. However we are sure there will be many questions and suggestions. Please feel free to email us and we will do our best to respond promptly and correctly. Have fun with Copper !!

Copper Grammar for Specification Files

Comments are C/C++ style

comment : /* ... */ | // ...

Basics

identifier : lower_id | upper_id | dummy_var

int_constant : integer constant

string_literal : string literal

lower_id : identifier beginning with lower-case letter

upper_id : identifier beginning with upper-case letter

dummy_var : dummy variable beginning with a "$"

Expressions

primary_expression
: identifier
| int_constant
| string_literal
| "(" expression ")"

postfix_expression
: primary_expression
| postfix_expression "[" expression "]"
| postfix_expression "(" ")"
| postfix_expression "(" argument_expression_list ")"
| postfix_expression "." identifier
| postfix_expression "->" identifier
| postfix_expression "++"
| postfix_expression "--"

argument_expression_list
: assignment_expression
| argument_expression_list "," assignment_expression

```
unary_expression
    : postfix_expression
    | "++" unary_expression
    | "--" unary_expression
    | unary_operator cast_expression
    | "sizeof" unary_expression
    | "sizeof" "(" type_name ")"

unary_operator  : "&" | "*" | "+" | "-" | "~" | "!"

cast_expression
    : unary_expression
    | "(" type_name ")" cast_expression
    | "(" type_name ")" list_initializer

multiplicative_expression
    : cast_expression
    | multiplicative_expression "*" cast_expression
    | multiplicative_expression "/" cast_expression
    | multiplicative_expression "%" cast_expression

additive_expression
    : multiplicative_expression
    | additive_expression "+" multiplicative_expression
    | additive_expression "-" multiplicative_expression

shift_expression
    : additive_expression
    | shift_expression "<<" additive_expression
    | shift_expression ">>" additive_expression

relational_expression
    : shift_expression { $$ = $1; }
    | relational_expression "<" shift_expression
    | relational_expression ">" shift_expression
    | relational_expression "<=" shift_expression
    | relational_expression ">=" shift_expression

equality_expression
    : relational_expression
    | equality_expression "==" relational_expression
    | equality_expression "!=" relational_expression

and_expression
    : equality_expression
    | and_expression "&" equality_expression

exclusive_or_expression
    : and_expression
    | exclusive_or_expression "^" and_expression

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression "|" exclusive_or_expression

logical_and_expression
    : inclusive_or_expression
```

| logical_and_expression "&&" inclusive_or_expression

logical_or_expression
: logical_and_expression
| logical_or_expression "||" logical_and_expression

conditional_expression
: logical_or_expression
| logical_or_expression "?" ":" conditional_expression
| logical_or_expression "?" expression ":" conditional_expression

assignment_expression
: conditional_expression
| unary_expression assignment_operator assignment_expression

assignment_operator : "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" | "^=" | "|="

expression
: assignment_expression
| expression "," assignment_expression

constant_expression : conditional_expression

Additional Information About C Procedures

procedure_name : postfix_expression

component
/*single name*/
: procedure_name
/*two names - second name to be substituted - ignore this for now*/
| procedure_name "|" procedure_name

component_list
/*name*/
: component
/*list*/
| component_list "," component

inline_list
/*name*/
: procedure_name
/*list*/
| inline_list "," procedure_name

program_info
: "program" component_list "{" program_decls "}"
| "program" component_list "{" "}"

program_decls
: program_decl
| program_decls program_decl

program_decl
: "specification" identifier "," "{" argument_expression_list "}" "," process_id ";"
| "specification" identifier "," "{" argument_expression_list "}" "," ltl_formula_name ";"

```
procedure_info
   : "procedure" procedure_name "{" procedure_decls "}"
   | "procedure" procedure_name "{" "}"

procedure_decls
   /*pred*/
   : predicate_decl
   /*pred_list*/
   | procedure_decls predicate_decl
   /*fair_loop*/
   | fair_loop_decl
   /*fair_loop_list*/
   | procedure_decls fair_loop_decl
   /*aux*/
   | auxiliary_decl
   /*aux_list*/
   | procedure_decls auxiliary_decl
   /*inline*/
   | inline_decl
   /*inline_list*/
   | procedure_decls inline_decl
   /*alias*/
   | alias_decl
   /*alias_list*/
   | procedure_decls alias_decl
   /*abstract*/
   | abstract_decl
   /*abstract_list*/
   | procedure_decls abstract_decl
   /*context*/
   | context_decl
   /*context_list*/
   | procedure_decls context_decl

predicate_decl : "predicate" predicate_list ";"

fair_loop_decl : "fair_loop" predicate_list ";"

auxiliary_decl : "auxiliary" predicate_list ";"

predicate_list
   /*pred*/
   : conditional_expression
   /*list*/
   | predicate_list "," conditional_expression

inline_decl : "inline" inline_list ";"

alias_decl : "alias" alias_item_list ";"

alias_item_list
   : alias_item
   | alias_item_list "," alias_item

alias_item
   : "{" conditional_expression "," points_to_list "}"
```

```
points_to_list
   /*name*/
   : conditional_expression
   /*list*/
   | points_to_list "," conditional_expression

abstract_decl
   : "abstract" abstract_item_list ";"

abstract_item_list
   : abstract_item
   | abstract_item_list "," abstract_item

abstract_item
   : "{" conditional_expression "," process_id "}"

context_decl : "context" int_constant ";"
```

FSP Productions

```
process_id : upper_id

action_label
   /*id*/
   : lower_id
   /*normal return*/
   | "return" "{" conditional_expression "}"
   /*void return*/
   | "return" "{" "}"
   /*assign*/
   | "{" conditional_expression  "=" "[" conditional_expression "]" "}"
   /*broadcast*/
   | lower_id "!!" "[" predicate_list "]"
   /*void send*/
   | lower_id "!" "[" "]"
   /*send*/
   | lower_id "!" "[" predicate_list "]"
   /*void receive*/
   | lower_id "?" "[" "]"
   /*receive*/
   | lower_id "?" "[" predicate_list "]"

action_label_list
   : action_label
   | action_label_list "," action_label

action_labels
   /*action*/
   : action_label

process_definition
   : process_id "=" process_body "."
   | process_id "=" process_body "+" "{" action_label_list "}" "."

process_body
   /*local*/
```

```
    : local_process
    /*list*/
    | local_process "," local_process_defs

local_process_defs
    /*local*/
    : local_process_def
    /*list*/
    | local_process_defs "," local_process_def

local_process_def : process_id "=" local_process

local_process
    /*stop*/
    : "STOP"
    /*id*/
    | process_id
    /*choice*/
    | "(" process_choice ")"

process_choice
    /*action*/
    : action_prefix
    /*choice*/
    | process_choice "|" action_prefix

action_prefix : prefix_actions PTR_OP local_process

prefix_actions
    /*action*/
    : action_labels
    /*prefix*/
    | prefix_actions "->" action_labels

prop_label
    : process_id "=" "{" predicate_list "}" "{" predicate_list "}" ";"
    | process_id "=" "{" predicate_list "}" "{" "}" ";"
    | process_id "=" "{" "}" "{" predicate_list "}" ";"

fsp_definition
    /*process*/
    : process_definition
    /*propositional_labeling*/
    | prop_label
```

LTL Productions

```
ltl_formula_name : upper_id primary_ltl_formula
    : "[" conditional_expression "]"
    | action_label
    | "(" ltl_formula ")"

unary_ltl_formula
    : primary_ltl_formula
    | "!" unary_ltl_formula
    | "#X" unary_ltl_formula
    | "#G" unary_ltl_formula
```

```
   | "#F" unary_ltl_formula

and_ltl_formula
   : unary_ltl_formula
   | and_ltl_formula "&" unary_ltl_formula

or_ltl_formula
   : and_ltl_formula
   | or_ltl_formula "|" and_ltl_formula

until_ltl_formula
   : or_ltl_formula
   | until_ltl_formula "#U" or_ltl_formula

release_ltl_formula
   : until_ltl_formula
   | release_ltl_formula "#R" until_ltl_formula

implies_ltl_formula
   : release_ltl_formula
   | release_ltl_formula "=>" implies_ltl_formula

ltl_formula : implies_ltl_formula

ltl_formula_def : "ltl" ltl_formula_name "{" ltl_formula ";" "}"
```

Top-Level Specification Productions

```
ext_def_list
   /*fsp*/
   : fsp_definition
   /*list_fsp*/
   | ext_def_list fsp_definition
   /*ltl*/
   | ltl_formula_def
   /*list_ltl*/
   | ext_def_list ltl_formula_def
   /*program*/
   | program_info
   /*list_program*/
   | ext_def_list program_info
   /*procedure*/
   | procedure_info
   /*list_procedure*/
   | ext_def_list procedure_info

spec_translation_unit : ext_def_list
```