cybersecurity centre

GCHQ Academic Centre of Excellence EPSRC

UNIVERSITY OF Southampton
Electronics and Computer Science

ESBMC: bounded model checking for C & C++

Denis A Nicole

2013-13-02

KAZAKH KBTU BRITISH TECHNICAL UNIVERSITY

cybersecurity centre

GCHQ Academic Centre of Excellence EPSRC

UNIVERSITY OF Southampton
Electronics and Computer Science

ESBMC is a Collaboration between

- University of Southampton
  - Jeremy Morse and Denis Nicole
- Federal University of Amazonas, Brazil
  - Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques and Lucas Cordeiro
- University of Stellenbosch, South Africa
  - Bernd Fischer

UNIVERSIDADE FEDERAL DO AMAZONAS
IN UNIVERSA SCIENTIA VERITAS

UNIVERSITY OF Southampton
Electronics and Computer Science

UNIVERSITEIT STELLENBOSCH UNIVERSITY

ESBMC is a *bounded* model checker

- It exhaustively analyses all possible behaviours of a (multithreaded) C or C++ program up to a *fixed* depth of
  - loop iteration (including backward jumps and recursion),
  - thread interleaving.
- Within these bounds, it checks for
  - C errors: pointer errors, arithmetic errors, array bounds, **`malloc()`**/**`free()`**, **`assert()`** failures, data races, etc.
  - Violation of *Linear Temporal Logic* specifications.

Model Checking is not Simulation

- *Simulation* (testing) checks correctness for a particular input and a particular thread interleaving.
- You need to run multiple simulations with different data and different timing before you get *some* assurance.
- *Model Checking* exhaustively analyses all possible behaviours over a range of possible inputs and generates a *witness*, a trace of program state, if there are any possible failures.
- Good-coverage simulation may be effective against "random" errors; it offers little protection against tailored attacks.

A change of viewpoint

- We used to think model checking would prove programs correct.
- Nowadays, we think of improving *test coverage:* performing a whole family of tests in one execution.
- And it gives an explicit failure *witness*.

The ESBMC approach

Originally based on CBMC.

1. Perform pre-processor substitution and C++ template expansion.
2. Simplify control flow to a *GOTO* form.
3. Constant fold.
4. Unwind program into a sequence of *single assignment* expressions.
5. Use an SMT solver (normally Z3) to verify the program.

The fundamental pattern

- We are trying to ensure that the *program semantics* imply the *assertions*.
- The program semantics is a conjunction of constraints which can be *assumed* from the program structure.
- So we seek to show:

$$assumptions \Rightarrow assertions$$

- We do so by looking for a *witness* to

$$\neg\,(assumptions \Rightarrow assertions) \quad \equiv$$
$$assumptions \wedge \neg\, assertions$$

A simple example

```
int i; int j=0;
for(i=1; i<4; i++) {
  j = j + i*i; }
assert (j == 14);
```

becomes

$$j\_0=0 \wedge i\_0=1 \wedge j\_1=j\_0+i\_0*i\_0 \wedge i\_1=2 \wedge j\_2=j\_1+i\_1*i\_1 \wedge i\_2=3 \wedge j\_3=j\_2+i\_2*i\_2 \wedge \neg\, j\_3=14$$

- Assignments in the program become *assumptions* relating the instantiations of the variables.
- Assertions and correctness conditions become *assertions* that must be true at various points in the execution.
- The SMT solver is asked to find a set of values for the unknowns which satisfies an expression which is the conjunction of all the assumptions and the inverse of the conjunction of the assertions.
- Such a set of values constitutes an explicit *witness* that the program is erroneous; this can be used for debugging.

C semantics are complicated

- Even this simple example is potentially wrong.
- The **`unsigned int`** is *guaranteed* not to overflow, but it is not a mathematical integer.
- The standard says it wraps modulo **`UINT_MAX`** + 1 and,
  `UINT_MAX` ≥ 65535
- Most sane compilers would have
  `UINT_MAX` = $2^{\text{WORD\_SIZE}}$ **-**1
- But if you made **`UINT_MAX`** = 65536, you would get a *field*…
  …which would enable some algebraic optimisations, e.g.

$$(a*4)/4 \quad \equiv \quad a$$

Sequencing

Sequenced before is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. (Conversely, if A is sequenced before B, then B is *sequenced after* A.) If A is not sequenced before or after B, then A and B are *unsequenced*.

Evaluations A and B are *indeterminately sequenced* when A is sequenced either before or after B, but it is unspecified which†) The presence of a sequence point between the evaluation of expressions A and B implies that every value computation and side effect associated with A is sequenced before every value computation and side effect associated with B.

† The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.

Implementation

The least requirements on a conforming implementation are:

- Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
- At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.
- The input and output dynamics of interactive devices shall take place as specified in 7.21.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

This is the observable behavior of the program.

Function calls

- An argument may be an expression of any complete object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.
- Wonderful: no mention of *indeterminately sequenced* or *unsequenced*. So, is this a correct program?

```
#include <assert.h>
void fun(int x, int y) {}
main() {
  int i=0;
  fun(i++, i++);
  assert (i == 2); }
```

Undefined behaviour

This, the only colour photo of the Trinity test was taken by my old housemate's (Iain Abey's) dad.

No, it isn't correct

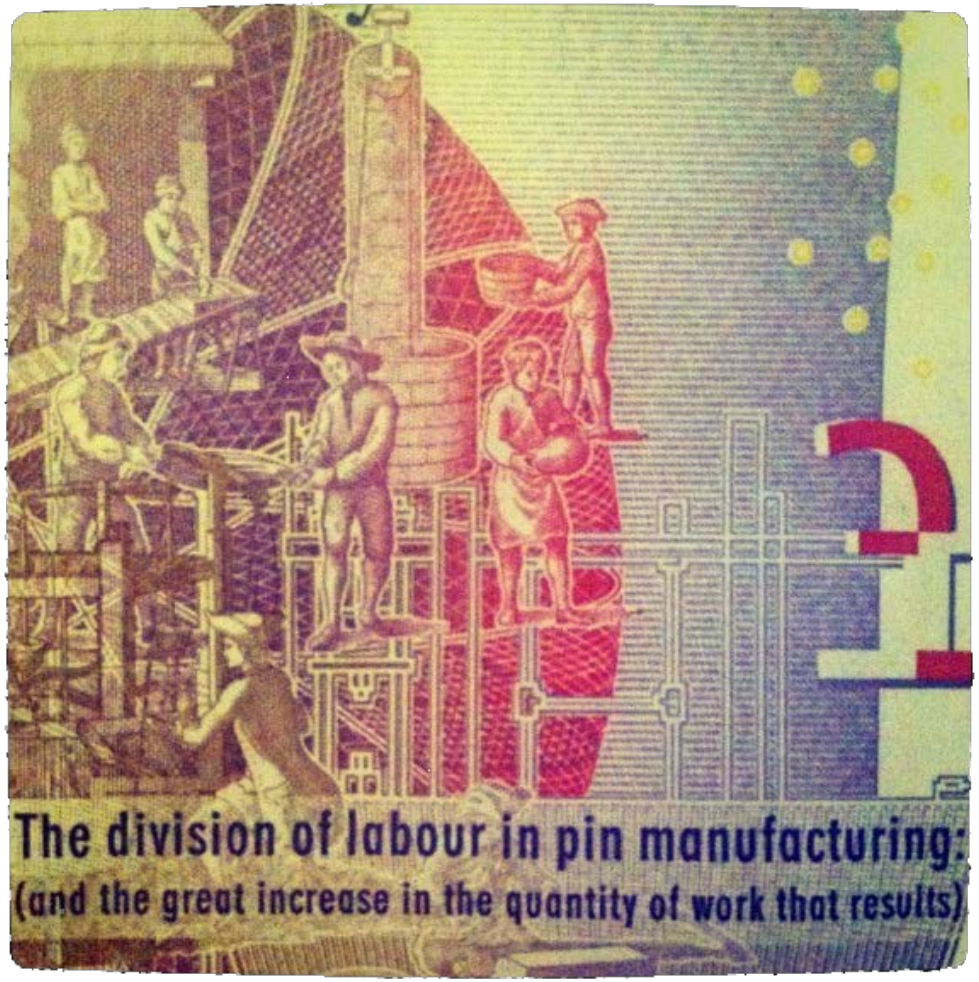
- Expressions (but not function calls) in function arguments *can* interleave:

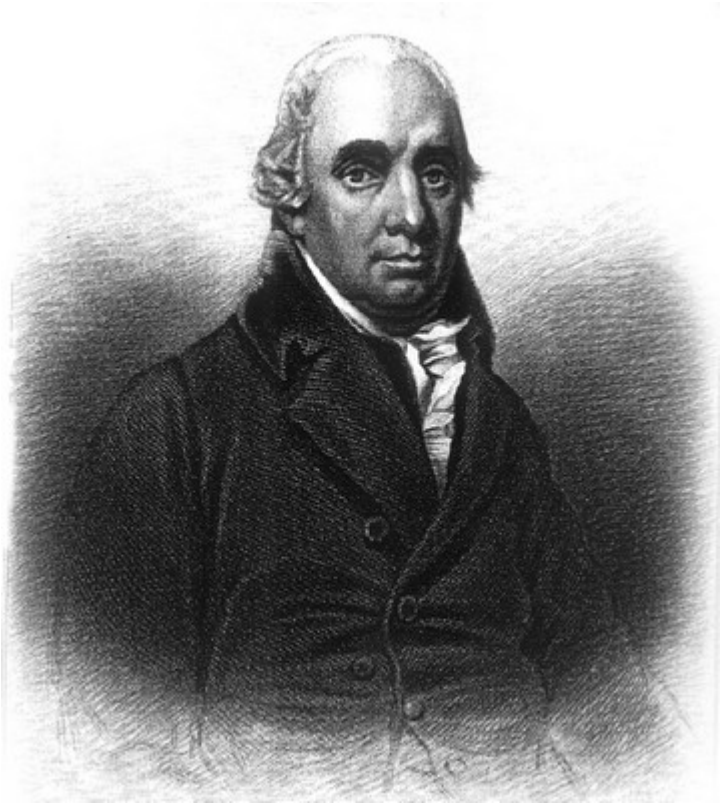
  The compiler may choose to perform the evaluation of expr1 before, after, or interleaved with the evaluation of expr2 . There are enough people who find this surprising that it comes up as a regular question on the newsgroups, but it's just a direct result of the C and C++ rules about sequence points.

  Herb Sutter
  More Exceptional C++: 40 New Engineering Puzzles,
  Item 20, part 1

Division of labour (Adam Smith)

- We use SMT solvers written elsewhere (Z3, Boolector)





SAT Solvers

- Have very good performance
- They seek any set of assignments to the free variables that will make a Boolean expression true, e.g.

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x_1$$

is satisfied by the *witness*

$$x_1 = x_2 = x_3 = \mathit{false}$$

- You don't get to choose the witness; the solver gives you the first it finds. But, if you really care, you can exclude this assignment and re-invoke the solver.

SMT solvers are a generalisation

- Linear arithmetic: multiplication is difficult!
- Uninterpreted functions

$$x_1 = y_1 \wedge x_2 = y_2 \Rightarrow f(x_1, x_2) = f(y_1, y_2)$$

- Array theory

$$\text{read}(\text{write}(A, i, x), i) = x$$

- And more: polynomials, bit fields...
- We can combine expressions

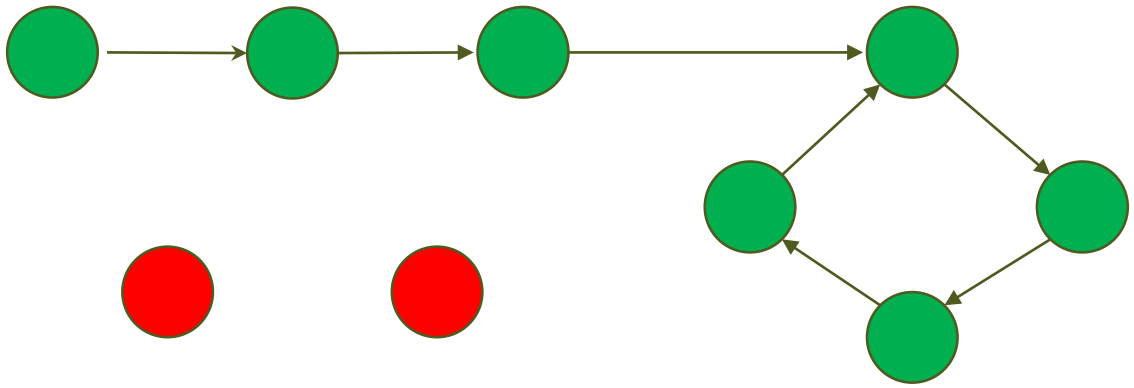
$$b + 2 = c \wedge f(\text{read}(\text{write}(a, b, 3), c - 2)) \neq f(c - b + 1)$$

Multithreaded programs

- In its normal mode of operation ESBMC generates separate SMT runs for all static interleavings of critical sections from different threads. This *assumes* the absence of data races.
- In an alternative mode, ESBMC can search for data races by exploring all interleavings between C statements. This is very compute-intensive. It is also wrong! C statements are in general not atomic. And modern reorderings and write pipelines make a nonsense of even the notion of interleaved execution.
- We are developing ESBMC to follow the new C11/C++11 memory model. Interleaving will then only be necessary at *barriers* associated with mutexes, atomic variables etc. Any potential race between barriers is an error, so far fewer interleaves need be considered.
- Note that there is *no* workable thread semantics before C11. (H-J Bohm, 2004)

Infinite programs

- We can use a variant of *k-induction* to check safety properties for infinitely executing (e.g. server) programs.
- This allows us to detect a program which repeats a set of reachable states.
- If such a repeat (or contraction) is found, then no new safety violations can arise.



Proof by induction: if we are lucky enough to be given a loop invariant

```
int i=0,j = 0;
while (i < 10000) {
  i++; j = j + i;
  assert(j == i*(i+1)/2); }
assert(j == 50005000);
```

Base case:

```
int j=0; i=0;
i++; j = j + i;
assert(j == i*(i+1)/2);
```

Finalisation:

```
int j, i=10000;
assume(j == i*(i+1)/2);
assert(j == 50005000);
```

Inductive step:

```
int i, j;
assume((i>0)&(i<10000)); //overflow?
assume(j == i*(i+1)/2);
i++; j= j + i;
assert(j == i*(i+1)/2);
```

Linear Temporal Logic properties

- Predicates on global (**`volatile`**) variables can be used to construct a *Linear Temporal Logic* specification for the program.
- The LTL can include both safety properties (such as class invariants) and liveness properties.
- In the case of infinite programs, bounded unwinding is unlikely to prove an LTL specification valid. It can find errors, and it can signal the validity of the LTL expression under various future behaviours.
- This capability is enabled by using a *monitor thread*, with optimised interleaving, to track the states of a nondeterministic Büchi automaton.

Propositional constants.

$$[w \models \mathsf{true}]_\omega = \top \qquad [w \models \mathsf{false}]_\omega = \bot \qquad [w \models p]_\omega = \begin{cases} \top \text{ iff } p \in w_0 \\ \bot \text{ iff } p \notin w_0 \end{cases}$$

Propositional operators.

$$[w \models \varphi \vee \psi]_\omega = [w \models \varphi]_\omega \sqcup [w \models \psi]_\omega \qquad [w \models \neg\varphi]_\omega = \overline{[w \models \varphi]_\omega}$$

Temporal operators.

$$[w \models \mathsf{X}\varphi]_\omega = [w^1 \models \varphi]_\omega$$

$$[w \models \mathsf{F}\varphi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \varphi]_\omega = \top \text{ for some } i \geq 0 \\ \bot \text{ otherwise} \end{cases}$$

$$[w \models \mathsf{G}\varphi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \varphi]_\omega = \top \text{ for all } i \geq 0 \\ \bot \text{ otherwise} \end{cases}$$

$$[w \models \varphi \mathsf{U} \psi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \psi]_\omega = \top \text{ for some } i \geq 0 \\ \qquad \text{and } [w^j \models \varphi]_\omega = \top \text{ for all } 0 \leq j < i \\ \bot \text{ otherwise} \end{cases}$$

$$[w \models \varphi \mathsf{R} \psi]_\omega = \begin{cases} \top \text{ iff } [w^i \models \psi]_\omega = \top \text{ for all } i \geq 0 \\ \qquad \text{or } [w^i \models \varphi]_\omega = \top \text{ for some } i \geq 0 \\ \qquad\qquad \text{and } [w^j \models \psi]_\omega = \top \text{ for all } 0 \leq j \leq i \\ \bot \text{ otherwise} \end{cases}$$

LTL algebra

- There is an implicit overall prefix **A** ("over all possible futures"), if you want to read the LTL as CTL*.
- Only **U** and **X** are fundamental

$$\mathbf{F}\ \psi \equiv \text{true}\ \mathbf{U}\ \psi$$

$$\mathbf{G}\ \psi \equiv \neg \mathbf{F} \neg\ \psi$$

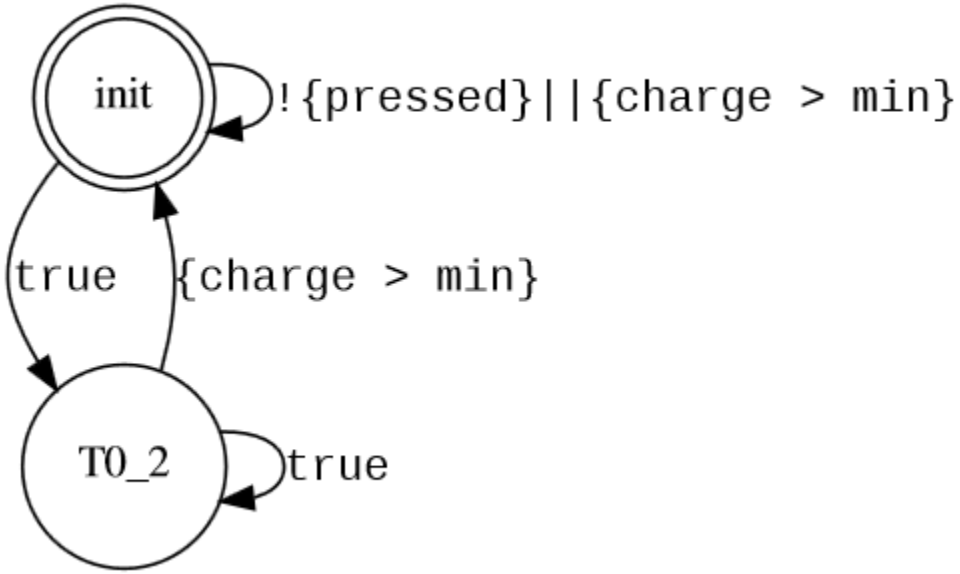
$$\varphi\ \mathbf{R}\ \psi \equiv \neg\ \varphi\ \mathbf{U} \neg\ \psi$$

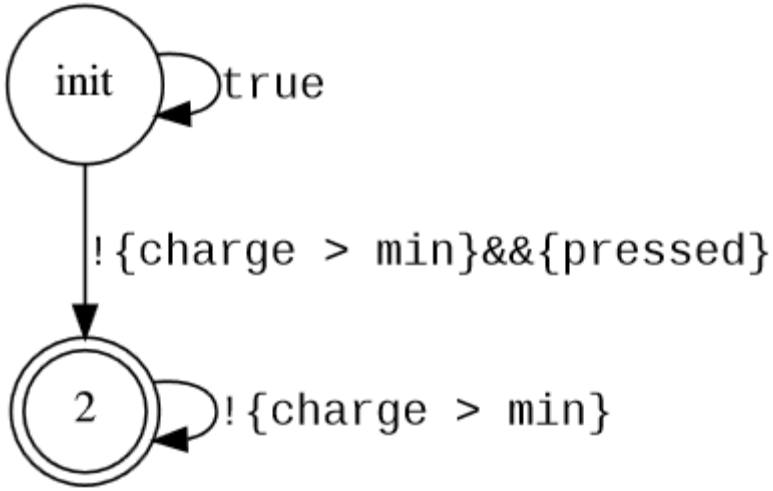
- Fixed points: unwinding into a state machine (Büchi Automaton)

$$\varphi\ \mathbf{U}\ \psi \equiv \psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))$$

The **U** operator is expressively complete with respect to first-order temporal properties on continuous, strictly linear temporal orders (J A W Kamp, 1968).

Büchi Automata





`G({pressed}->F{charge>min})`

`!G({pressed}->F{charge>min})`

The (nondeterministic) automaton *accepts* if it can visit an accepting state infinitely often.

Stutter-independent bounded trace semantics

The bounded trace semantics *of LTL formulas is given by*

$$[u \models \varphi]_B = \begin{cases} \top & \text{iff } \forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top \\ \top^p & \text{iff } [uu_{n-1}^\omega \models \varphi]_\omega = \top \wedge \exists w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \bot \\ \bot^p & \text{iff } [uu_{n-1}^\omega \models \varphi]_\omega = \bot \wedge \exists w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \top \\ \bot & \text{iff } \forall w \in \Sigma^\omega \cdot [uw \models \varphi]_\omega = \bot \end{cases}$$

for a finite trace $u \in \Sigma^*$ *of length* $n > 0$ *and an LTL formula* φ.

Stutter-independence (Lamport) is typical of software verification. In contrast to hardware verification, the clock tick has little significance, so we expect software LTL specifications to be writable without the **X** (next) operator. Indeed, the **X** operator can be eliminated from any stutter-independent formula.

LTL expressions

- **Safety** e.g. a *class invariant* which must hold (almost) everywhere, in contrast to a *point* `assert(P)` which only has to hold at one point in the execution.

  `G{P}`

  or

  `G{outside_instance_methods & P}`

- **Co-safety** or *convergence*. Something must eventually happen.

  `F{P}`

- **True liveness.** Events always get a response

  `G({P}->F{Q})`

True liveness example

```
unsigned int i = 0;
int main() {
  while(1)
    i++; }
```

LTL specification:

```
G({i%2==0} -> F{i%3==0})
```

As we increase the unwind bound from 1 to 12, ESBMC reports:

$$\top^p, \bot^p, \top^p, \bot^p, \bot^p, \top^p, \top^p, \bot^p, \top^p, \bot^p, \bot^p, \top^p$$

Red herrings

- We only consider infinite (ω-language) traces.
- We avoid the *weak until*
  $\text{true}\ \mathsf{U}_{\mathsf{W}}\ \psi \equiv \text{true}$
- We avoid the *strict future until* (stutter confusion)
  $\text{false}\ \mathsf{U}^{>}\ \psi \equiv \mathsf{X}\ \psi$
- We don't need *past time operators:* our programs start but do not finish: we have infinite stutter extension.
- A nondeterministic Büchi automaton is stronger than LTL (it does S1S), but why should we care? Our C *monitor* can calculate and track any observable state, if necessary.

Partial loops

- What do we do if we run out of loop iterations?
  - Give up?
  - Press on regardless?
- In ESBMC it's the user's choice.
- If we "press on regardless", we have completed a number of whole loop iterations.
- So, we respect the *loop invariant* but not the *termination condition*.

More partial loops

- Very rarely, it might be better to keep the termination condition instead:

```
int j = 0;
for ( i=0; i<= 10000; i++) {
  j = j + i*i; }
assert(i == 10001);
```

- Future work: we could keep both loop invariant and termination condition, but relax some other assumptions.

Improved performance in 2012

- **State hashing**
  We detect when different interleaves result in the same set of reachable states and merge the models.
- **Monotonic partial order reduction**
  Optimal reduction in the number of interleavings.
- **k-induction**
- **LTL specifications**

Improved performance in 2013

- The old CBMC string-based program intermediate representation has been replaced with a C++ class hierarchy, resulting in a 200% performance improvement.
- To avoid very large SMT expressions which calculate possible indexes into structures, we extended a static pointer analysis to determine the weakest alignment guarantee that a particular pointer variable provides, and inserted padding in structures to make all fields align to word boundaries.
- In cases where our static analysis does not allow us to determine loop bounds, we now follow an iterative scheme, running the SMT solver on increasing unwinds until it determines that the loop must have terminated.

C++

- A major set of extensions to ESBMC allows us to support C++, including templates and the containers of the *standard template library*.
- We are able to manipulate containers efficiently via a direct model of the container properties, without having to expand a specific implementation of the container library.

Improvement by competition

- The field of C model checking research is now large enough to support annual competitions; perhaps the best known is that held in conjunction with the *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS).
- The team is proud to report that ESBMC v1.17 won the Gold Medal in the *SystemC* and *Concurrency* categories and the Bronze Medal in the overall ranking of the first *International Competition on Software Verification* at TACAS 2012.
- ESBMC v1.20 won the Bronze Medal in the overall ranking of the second competition at TACAS 2013.
- Our entry for 2014 has just been submitted.