Comparing Verification Condition Generation with Symbolic Execution

… with an Introduction to Syxc

Z3 SIG Meeting
2nd to 4th November 2011, Cambridge

Malte Schwerhoff, ETH Zürich
Joint work with Yannis Kassios, Peter Müller

Outline

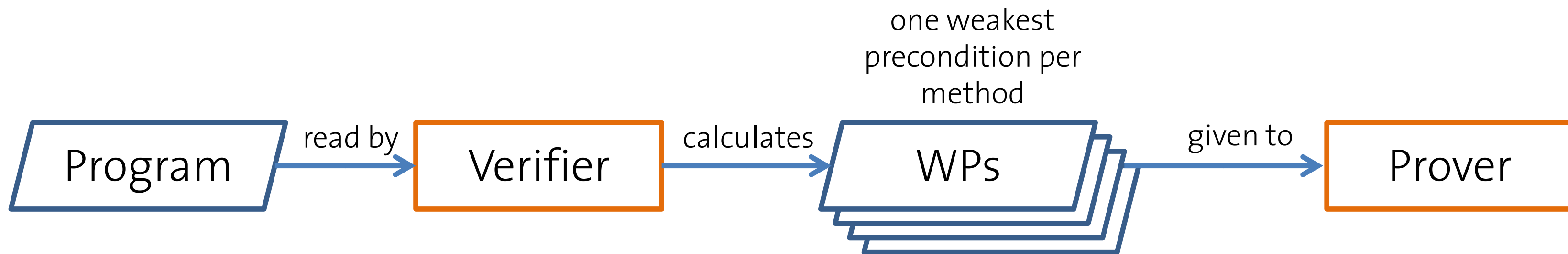
1. Background
2. Verifiers
3. Experiment

Background

Chalice

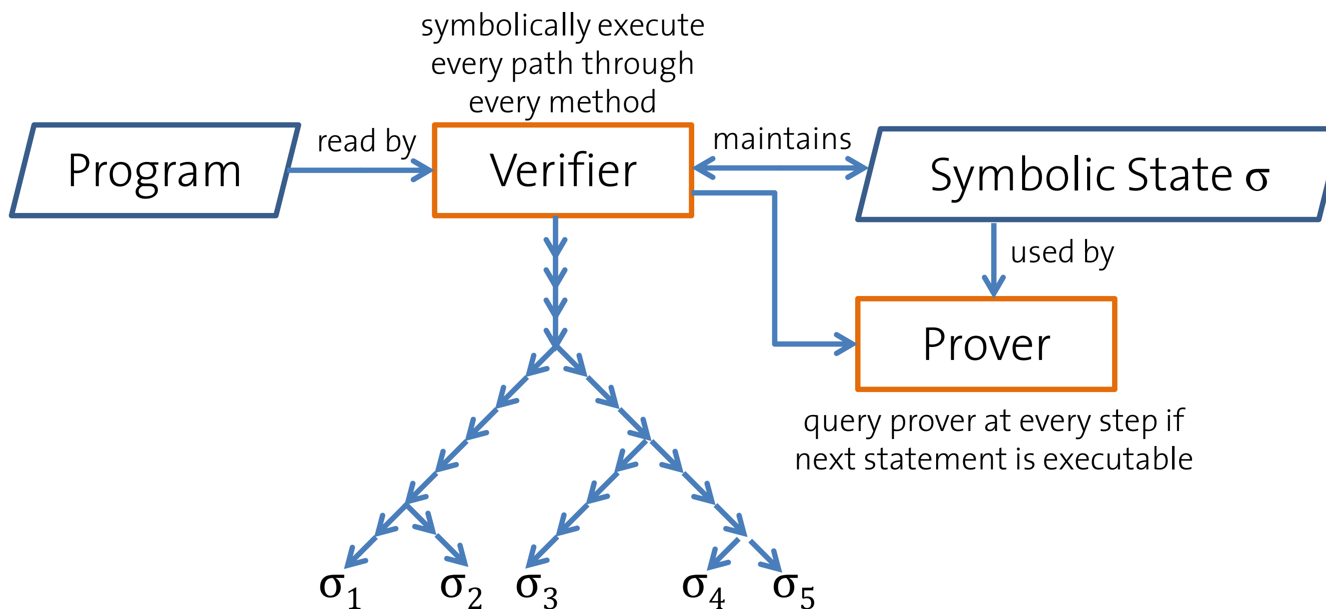
- Microsoft research language by Rustan Leino and others
- User provided specifications
  - Pre- and postconditions, loop invariants
  - *Pure functions* for information hiding (getters) and specification reasons
  - *Abstract predicates* for information hiding and to specify recursive structures
- Reasoning about side effects with *implicit dynamic frames*
- Fork-join concurrency with static dead-lock detection
- Shared data due to *fractional permissions,* monitors and locks
- Asynchronous communication via message passing channels

Verification condition generation vs. symbolic execution



VCG

→ Query prover once per method with all available information



SE

→ Query prover often with limited information

Verifiers

(well, mainly one)

VCG-based Chalice verifier: Chalice

- Encodes a Chalice program in the *intermediate verification language* Boogie
- Heaps and permissions are encoded in Boogie as updateable maps
- Boogie verifier computes weakest preconditions and uses Z3 to discharge proof obligations
- User functions are axiomatised, i.e. for all heaps, permissions maps, and arguments, a function application is equivalent to evaluating its body w.r.t. the given heap, permissions and arguments
- Chalice is implemented in Scala

Chalice: Example verification

```
class Cell {
  var x: int

  function get(): int
    requires rd(x)
  { x }
}
```

```
method set(y: int)
  requires acc(x)
  ensures acc(x) && get() == y
{ x := y }
```

Chalice: Example verification

```
class Cell {
  var x: int

  function get(): int
    requires rd(x)
  { x }
}

method set(y: int)
  requires acc(x)
  ensures acc(x) && get() == y
{ x := y }
```

the formula finally given to Z3

```
method client(c: Cell, y: int)
  requires acc(c.x)
  ensures acc(c.x) && c.get() == y + 1
{
      // Π(c,x)=100 ⇒ (Π(c,x)=100  ∧ ∀ u • (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦u],Π,c)=y
      // ⇒ (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦H(c,x)+1][(c,x)↦u],Π,c)=y+1)))

      // Π(c,x)=100 ∧ (∀ u • (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦u],Π,c)=y
      // ⇒ (Π(c,x)=100 ∧ Cell.get(H[(c,x)↦H(c,x)+1][(c,x)↦u],Π,c)=y+1)))
  call c.set(y)

      // Π(c,x)=100 ∧ Cell.get(H[(c,x)↦H(c,x)+1],Π,c)=y+1
  c.x := c.x + 1

      // Π(c,x)=100 ∧ Cell.get(H,Π,c)=y+1
}
```

- Weakest precondition is computed bottom-up, starting with the user-provided postcondition
- Weakest precondition must be implied by the user-provided precondition

SE-based Chalice verifier: Syxc

- VeriCool-style symbolic execution with a *symbolic state* comprising:
  - A *store* γ mapping local variables to symbolic values (*terms*)
  - Symbolic *heaps* h and g (old heap) consisting of *heap chunks*, i.e. quadruples

    $r.f \mapsto v \ \# \ p$

    representing

    "field $r.f$ has the value v and we have p permissions to the field"
  - *Path conditions* π with assumptions such as $v > 0,\ r \neq \mathrm{null}$
- Syxc is written in Scala

Syxc

- Store and heap are maintained by Syxc, i.e. as Scala data structures, path conditions are managed by Z3's push-pop scopes:
  - Knowledge is partitioned, Z3 does not know about heaps and permissions
  - Idea: Relieving Z3 from reasoning about the heap increases performance
- Functions are not axiomatised
  - Equality between function application and body evaluation is added when a function application is evaluated
  - Necessary, since heap and path conditions are separated, but functions are still framed by their heap footprint

Syxc: Example verification

```
class Cell {
  var x: int

  function get(): int
    requires rd(x)
  { x }
}
```

```
method set(y: int)
  requires acc(x)
  ensures acc(x) && get() == y
{ x := y }
```

```
method client(c: Cell, y: int)
  requires acc(c.x)
{
      // γ: {c ↦ tc, y ↦ ty}
      // h: {tc.x ↦ tx # 100}
      // π: {}
  call c.set(y)
      // h: {tc.x ↦ tx' # 100}
      // π: {Cell.get(tc, tx') == ty ∧ Cell.get(tc, tx') == tx'}
  c.x := c.x + 1
      // h: {tc.x ↦ tx' + 1 # 100}
  assert acc(c.x) && c.get() == y + 1
      // π: {Cell.get(tc, tx') == ty ∧ Cell.get(tc, tx') == tx,
      //      ∧ Cell.get(tc, tx' + 1) == tx' + 1}
}
```

Z3

Z3

Z3

Z3

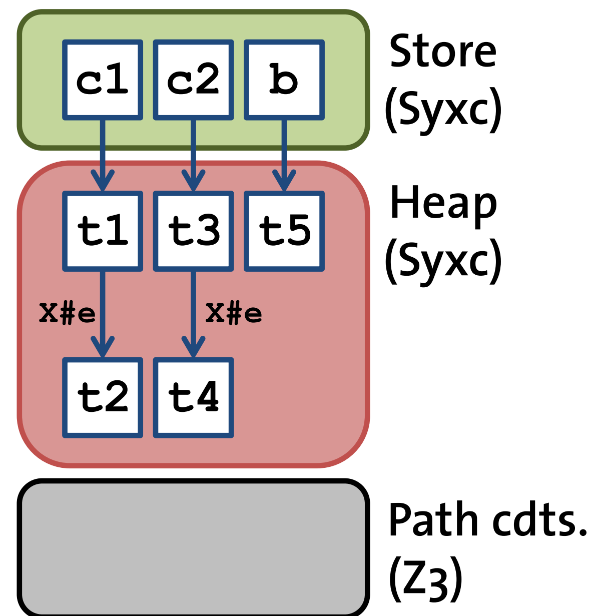
- Symbolic execution proceeds top-down
- Z3 is invoked with the current π to discharge proof obligations arising from preconditions and assertions

Syxc: State separation consequence I

- Separation between heap and path conditions has consequences

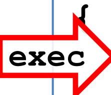
exec

```
method client(c1: Cell, c2: Cell, b: bool)
  requires rd(c1.x) && rd(c2.x)
{
  if (c1 == c2) {
    assert c1.x == c2.x


  }
}
```

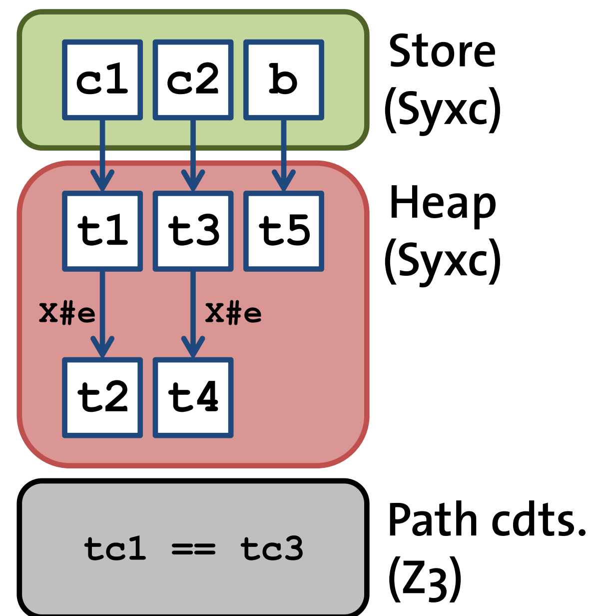


Syxc: State separation consequence I

- Separation between heap and path conditions has consequences



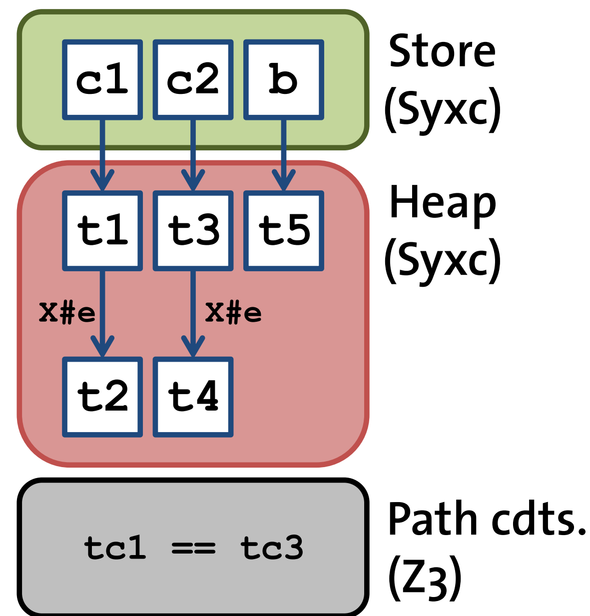

```
method client(c1: Cell, c2: Cell, b: bool)
  requires rd(c1.x) && rd(c2.x)
{
  if (c1 == c2) {
    assert c1.x == c2.x



  }
}
```



Syxc: State separation consequence I

- Separation between heap and path conditions has consequences

```
method client(c1: Cell, c2: Cell, b: bool)
  requires rd(c1.x) && rd(c2.x)
{
  if (c1 == c2) {
    assert c1.x == c2.x
        // Would fail naively because
        // tx1 = tx2 is unknown to Z3
  }
}
```



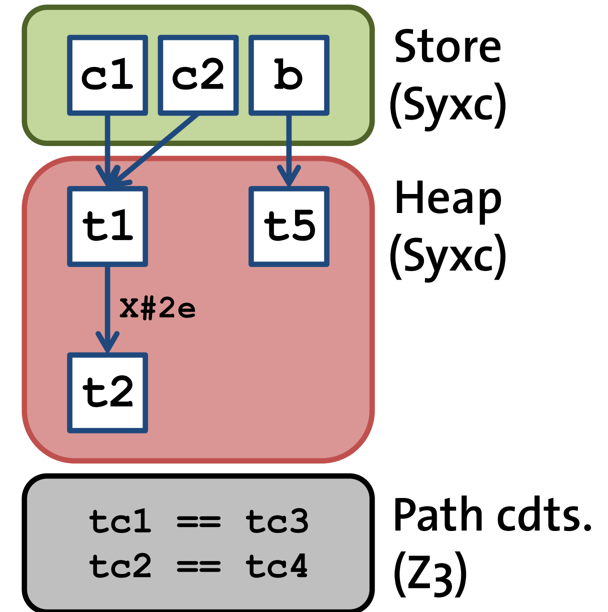
- Solution: *Heap compression*, that is, modifying the heap according to the current path conditions

Syxc: State separation consequence I

- Separation between heap and path conditions has consequences

```
method client(c1: Cell, c2: Cell, b: bool)
  requires rd(c1.x) && rd(c2.x)
{
  if (c1 == c2) {
    assert c1.x == c2.x
        // Holds


  }
}
```

exec



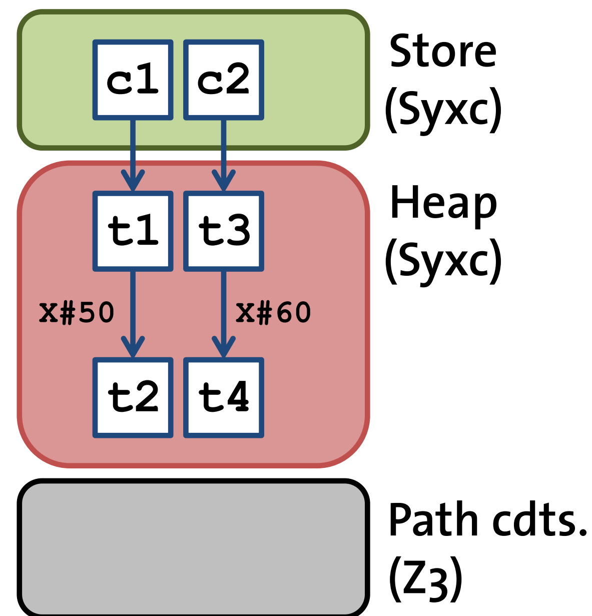
- For all pairs t, t' of receivers and all fields f in h, invoke Z3 to check if t = t'; if so, merge chunks
- Expensive operation: $\mathcal{O}(n^2)$
- Worse, newly added equalities require iterative proceeding: $\mathcal{O}(n^3)$

Syxc: State separation consequence II

- Separation between heap and path conditions has consequences

exec

```
method client(c1: Cell, c2: Cell)
  requires acc(c1.x, 50) && acc(c2.x, 60)
{
  assert c1 != c2


}
```

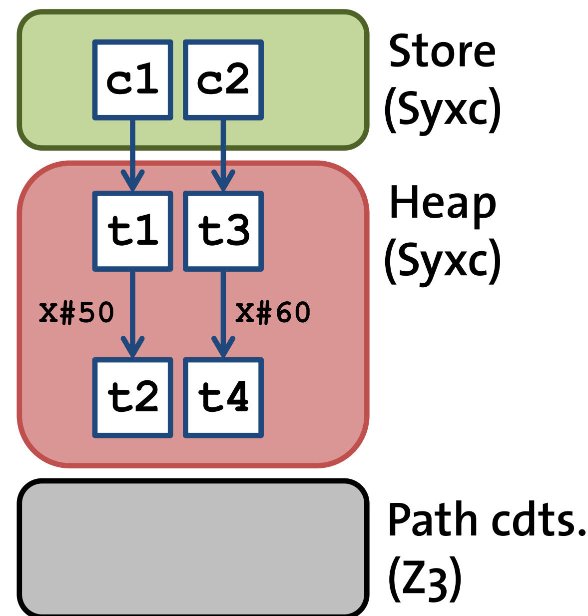


Syxc: State separation consequence II

- Separation between heap and path conditions has consequences

```
method client(c1: Cell, c2: Cell)
  requires acc(c1.x, 50) && acc(c2.x, 60)
{
  assert c1 != c2
      // Would fail naively because Z3 does
      // not know about permissions (at most 100)
}
```

exec



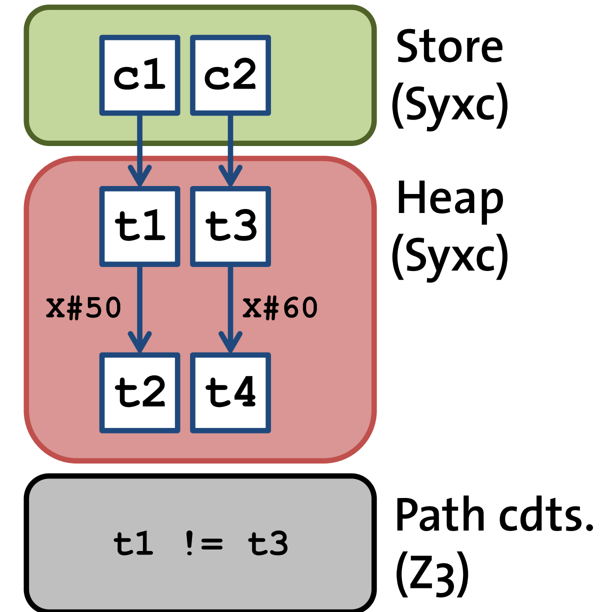
- Solution: Compute *object disjointness* based on field names and permissions:
  - For all pairs of chunks with receivers t, t' and the same field id f, invoke Z3 to check if the combined permissions exceed 100
  - If so, add assumption t ≠ t' to the path conditions
  - Expensive operation: $\mathcal{O}(n^2)$

Syxc: State separation consequence II

- Separation between heap and path conditions has consequences

exec

```
method client(c1: Cell, c2: Cell)
  requires acc(c1.x, 50) && acc(c2.x, 60)
{
  assert c1 != c2
      // Holds

}
```



- Solution: Compute *object disjointness* based on field names and permissions:
  - For all pairs of chunks with receivers t, t' and the same field id f, invoke Z3 to check if the combined permissions exceed 100
  - If so, add assumption $t \neq t'$ to the path conditions
  - Expensive operation: $\mathcal{O}(n^2)$

Syxc: Branching

- Symbolic execution branches on
  - If-then-else statements
  - If-then-else expressions (ternary operator) and implications
- → possible explosion of the number of paths

```
method client(c: Cell, b: bool)
  requires b ==> acc(c.x)
{ ... }
```

- Two execution paths with different heaps (and path conditions)
  1. Heap h contains chunk $tc.x \mapsto tx \; \# \; 100$
  2. Heap h does not contain such a chunk
- Possible optimisation: *conditional chunks*
  - Single path with chunk $tc.x \mapsto tx \; \# \; (tb \; ?100 : 0)$
  - Prototypical implementation looked promising, but no systematic benchmarks have been run

Syxc: Branching

```
method client(b: bool, i: int)
  requires b ==> i > 0
{ ... }
```

- *Pure expression*, i.e. it has no effect on the heap
  - Caused heavy branching in an impl. of Peterson's Algorithm (4x slower)
- Since the expression is pure
  - Encode it as a Z3 implication $\mathrm{tb} \Rightarrow \mathrm{ti} > 0$
  - Add it to π, continue without branching
- Decreased verification time significantly; interpretations
  - Z3 is heavily optimised and branches faster than Syxc?
  - Additional knowledge allows Z3 to prune branches? (our assumption)

Experiment

Setup

- Benchmarked 22 tests from the Chalice test suite
- Tests existed already before Syxc had been developed
- Tests exercise main the Chalice features such as fractional permissions, objects, threads, locks, and message passing

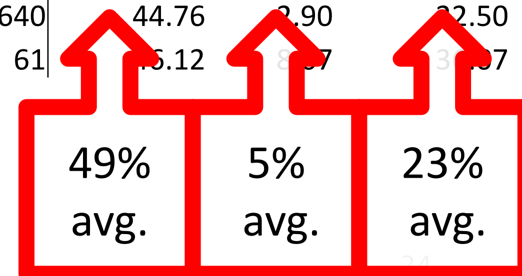
- Syxc uses Chalice (as a library) to parse input into its AST
- Z3 3.1, SMTLib2 frontend via std-io, interaction is logged in a file
- We ensured that Syxc and Chalice use Z3 in comparable settings, for example by using nearly identical Z3 configurations

- Averaging statistics over ten runs per tool and test case:
  - Verify each test case and record the runtime
  - Run Z3 on the created SMTLib2 logfile to get Z3 statistics

Results

File			Syxc			Chalice			Syxc rel. to Chalice		
Name	LOC	Meth.	Time (s)	QI	Conflicts	Time (s)	QI	Conflicts	Time (%)	QI (%)	Conflicts (%)
AVLTree.iterative	212	3	2.24	231	177	9.45	146238	1288	23.70	0.16	13.72
AVLTree.nokeys	572	19	34.21	2357	4304	154.03	2541875	19593	22.21	0.09	21.97
Cell	131	11	1.78	233	40	3.23	9599	526	55.11	2.43	7.60
CopyLessMessagePassing	54	3	1.46	218	35	2.72	4769	210	53.68	4.57	16.67
CopyLessMessagePassing-w/a	62	3	1.71	751	75	2.76	4014	181	61.96	18.71	41.44
CopyLessMessagePassing-w/a2	66	3	1.66	792	78	2.92	10542	325	56.85	7.51	24.00
Dining-philosophers	74	3	1.62	1181	128	3.09	17144	416	52.43	6.89	30.82
Iterator	123	6	3.65	436	210	3.27	7577	316	111.62	5.75	66.39
Iterator2	111	6	1.69	174	41	3.17	12867	304	53.31	1.35	13.49
LoopLockChange	69	5	1.26	220	42	2.60	2624	207	48.46	8.38	20.29
OwickiGries	31	2	1.21	121	41	2.61	5696	206	46.36	2.12	19.90
PetersonsAlgorithm	70	3	2.12	1196	278	7.36	160574	1500	28.80	0.74	18.53
ProdConsChannel	78	6	1.43	282	117	2.78	4640	316	51.44	6.08	37.03
Producer-consumer	171	11	1.91	466	147	3.84	17667	598	49.74	2.64	24.58
Quantifiers	28	1	1.02	16	4	2.23	196	12	45.74	8.16	33.33
RockBand	109	13	1.30	79	14	2.92	9129	341	44.52	0.87	4.11
Sieve	56	4	1.42	308	93	2.74	7596	246	51.82	4.05	37.80
Swap	19	2	0.98	10	2	2.20	261	20	44.55	3.83	10.00
Prog1	33	4	1.07	40	27	2.32	1101	106	46.12	3.63	25.47
Prog2	52	7	1.11	22	17	2.31	258	28	48.05	8.53	60.71
Prog3	163	16	1.58	627	144	3.53	21632	640	44.76	2.90	22.50
Prog4	19	1	1.07	76	22	2.32	877	61	[illegible]	[illegible]	[illegible]

LOC = Lines of code
Meth. = Methods
Time = Verification time (sec)
QI = Quantifier instantiations
Conflicts ≅ Explored search space



Results

File			Syxc			Chalice			Syxc rel. to Chalice		
Name	LOC	Meth.	Time (s)	QI	Conflicts	Time (s)	QI	Conflicts	Time (%)	QI (%)	Conflicts (%)
AVLTree.iterative	212	3	2.24	231	177	9.45	146238	1288	23.70	0.16	13.72
AVLTree.nokeys	572	19	34.21	2357	4304	154.03	2541875	19593	22.21	0.09	21.97
Cell	131	11	1.78	233	40	3.23	9599	526	55.11	2.43	7.60
CopyLessMessagePassing	54	3	1.46	218	35	2.72	4769	210	53.68	4.57	16.67
CopyLessMessagePassing-w/a	62	3	1.71	751	75	2.76	4014	181	61.96	18.71	41.44
CopyLessMessagePassing-w/a2	66	3	1.66	792	78	2.92	10542	325	56.85	7.51	24.00
Dining-philosophers	74	3	1.62	1181	128	3.09	17144	416	52.43	6.89	30.82
Iterator	123	6	3.65	436	210	3.27	7577	316	111.62	5.75	66.39
Iterator2	111	6	1.69	174	41	3.17	12867	304	53.31	1.35	13.49
LoopLockChange	69	5	1.26	220	42	2.60	2624	207	48.46	8.38	20.29
OwickiGries	31	2	1.21	121	41	2.61	5696	206	46.36	2.12	19.90
PetersonsAlgorithm	70	3	2.12	1196	278	7.36	160574	1500	28.80	0.74	18.53
ProdConsChannel	78	6	1.43	282	117	2.78	4640	316	51.44	6.08	37.03
Producer-consumer	171	11	1.91	466	147	3.84	17667	598	49.74	2.64	24.58
Quantifiers	28	1	1.02	16	4	2.23	196	12	45.74	8.16	33.33
RockBand	109	13	1.30	79	14	2.92	9129	341	44.52	0.87	4.11
Sieve	56	4	1.42	308	93	2.74	7596	246	51.82	4.05	37.80
Swap	19	2	0.98	10	2	2.20	261	20	44.55	3.83	10.00
Prog1	33	4	1.07	40	27	2.32	1101	106	46.12	3.63	25.47
Prog2	52	7	1.11	22	17	2.31	258	28	48.05	8.53	60.71
Prog3	163	16	1.58	627	144	3.53	21632	640	44.76	2.90	22.50
Prog4	19	1	1.07	76	22	2.32	877	61	46.12	8.67	36.07

- Longest method (iterative); longest example (nokeys)
- Best runtime (24%) and quantifier instantiation (0.2%)

Results

File			Syxc			Chalice			Syxc rel. to Chalice		
Name	LOC	Meth.	Time (s)	QI	Conflicts	Time (s)	QI	Conflicts	Time (%)	QI (%)	Conflicts (%)
AVLTree.iterative	212	3	2.24	231	177	9.45	146238	1288	23.70	0.16	13.72
AVLTree.nokeys	572	19	34.21	2357	4304	154.03	2541875	19593	22.21	0.09	21.97
Cell	131	11	1.78	233	40	3.23	9599	526	55.11	2.43	7.60
CopyLessMessagePassing	54	3	1.46	218	35	2.72	4769	210	53.68	4.57	16.67
CopyLessMessagePassing-w/a	62	3	1.71	751	75	2.76	4014	181	61.96	18.71	41.44
CopyLessMessagePassing-w/a2	66	3	1.66	792	78	2.92	10542	325	56.85	7.51	24.00
Dining-philosophers	74	3	1.62	1181	128	3.09	17144	416	52.43	6.89	30.82
Iterator	123	6	3.65	436	210	3.27	7577	316	111.62	5.75	66.39
Iterator2	111	6	1.69	174	41	3.17	12867	304	53.31	1.35	13.49
LoopLockChange	69	5	1.26	220	42	2.60	2624	207	48.46	8.38	20.29
OwickiGries	31	2	1.21	121	41	2.61	5696	206	46.36	2.12	19.90
PetersonsAlgorithm	70	3	2.12	1196	278	7.36	160574	1500	28.80	0.74	18.53
ProdConsChannel	78	6	1.43	282	117	2.78	4640	316	51.44	6.08	37.03
Producer-consumer	171	11	1.91	466	147	3.84	17667	598	49.74	2.64	24.58
Quantifiers	28	1	1.02	16	4	2.23	196	12	45.74	8.16	33.33
RockBand	109	13	1.30	79	14	2.92	9129	341	44.52	0.87	4.11
Sieve	56	4	1.42	308	93	2.74	7596	246	51.82	4.05	37.80
Swap	19	2	0.98	10	2	2.20	261	20	44.55	3.83	10.00
Prog1	33	4	1.07	40	27	2.32	1101	106	46.12	3.63	25.47
Prog2	52	7	1.11	22	17	2.31	258	28	48.05	8.53	60.71
Prog3	163	16	1.58	627	144	3.53	21632	640	44.76	2.90	22.50
Prog4	19	1	1.07	76	22	2.32	877	61	46.12	8.67	36.07

- Similar behaviour for test case PetersonsAlgorithm: Runtime speedup above average, very low QI ratio

Results

File			Syxc			Chalice			Syxc rel. to Chalice		
Name	**LOC**	**Meth.**	**Time (s)**	**QI**	**Conflicts**	**Time (s)**	**QI**	**Conflicts**	**Time (%)**	**QI (%)**	**Conflicts (%)**
AVLTree.iterative	212	3	2.24	231	177	9.45	146238	1288	23.70	0.16	13.72
AVLTree.nokeys	572	19	34.21	2357	4304	154.03	2541875	19593	22.21	0.09	21.97
Cell	131	11	1.78	233	40	3.23	9599	526	55.11	2.43	7.60
CopyLessMessagePassing	54	3	1.46	218	35	2.72	4769	210	53.68	4.57	16.67
CopyLessMessagePassing-w/a	62	3	1.71	751	75	2.76	4014	181	61.96	18.71	41.44
CopyLessMessagePassing-w/a2	66	3	1.66	792	78	2.92	10542	325	56.85	7.51	24.00
Dining-philosophers	74	3	1.62	1181	128	3.09	17144	416	52.43	6.89	30.82
Iterator	123	6	3.65	436	210	3.27	7577	316	111.62	5.75	66.39
Iterator2	111	6	1.69	174	41	3.17	12867	304	53.31	1.35	13.49
LoopLockChange	69	5	1.26	220	42	2.60	2624	207	48.46	8.38	20.29
OwickiGries	31	2	1.21	121	41	2.61	5696	206	46.36	2.12	19.90
PetersonsAlgorithm	70	3	2.12	1196	278	7.36	160574	1500	28.80	0.74	18.53
ProdConsChannel	78	6	1.43	282	117	2.78	4640	316	51.44	6.08	37.03
Producer-consumer	171	11	1.91	466	147	3.84	17667	598	49.74	2.64	24.58
Quantifiers	28	1	1.02	16	4	2.23	196	12	45.74	8.16	33.33
RockBand	109	13	1.30	79	14	2.92	9129	341	44.52	0.87	4.11
Sieve	56	4	1.42	308	93	2.74	7596	246	51.82	4.05	37.80
Swap	19	2	0.98	10	2	2.20	261	20	44.55	3.83	10.00
Prog1	33	4	1.07	40	27	2.32	1101	106	46.12	3.63	25.47
Prog2	52	7	1.11	22	17	2.31	258	28	48.05	8.53	60.71
Prog3	163	16	1.58	627	144	3.53	21632	640	44.76	2.90	22.50
Prog4	19	1	1.07	76	22	2.32	877	61	46.12	8.67	36.07

- Conflicts are less conclusive:
  - Iterator: conflicts above average, average QI, negative speedup
  - Prog2: conflicts above average, QI even higher, average speedup

Conclusions

- Test suite is still rather small, results have to be taken with caution!
- Syxc requires in average only 5% quantifier instantiations
  $\longrightarrow$ indicates that SE is more predictive than VCG
- Syxc causes in average only 25% conflicts, but with greater variation
  $\longrightarrow$ indicates that SE-based verification is more focused
- We assume that the number of QI is significantly smaller due to state separation into heap and path conditions
  - Less information for the prover, less non-goal-directed work
  - Separation of labour between verifier and prover seems beneficial
- However, limited information might not always be beneficial, as hinted at by the branching problem

Future work

- Investigate outliers
- Evaluate different heap compression strategies
- Implement new permission model
- Implement and benchmark conditional chunks
- Parallelisation: Methods, branches, others?
- Responsiveness: Cache symbolic state to re-execute as little as possible?
- Much more difficult: a debugger on top of Syxc
  - How to present symbolic state?
  - Which information is relevant and helpful, which isn't?
  - Evaluate different approaches

Questions?

Backup Slides

Chalice: Example

```
class Cell {
  var x: int

  predicate P { acc(x) }

 function get(): int
    requires rd(P)
  { unfolding rd(P) in x }

  method set(y: int)
    requires P
    ensures P && get() == y
  {
    unfold P
    x := y
    fold P
  }

  method clone() returns (c: Cell)
    requires rd(P)
    ensures rd(P) && c.P && c.get() == get()
  {
    c := new Cell
    fold c.P
    call c.set(get())
  }
}
```

```
class Client {
  method fails(c: Cell)
    requires c.P
  {
    fork tk1 := c.set(1)
    fork tk2 := c.set(2) // ERR
  }

  method succeeds(c: Cell)
    requires c.P
  {
    fork tk1 := c.clone()
    fork tk2 := c.clone()
  }
}
```

Syxc: Branching

```
method client(c: Cell, b: bool)
  requires b ==> acc(c.x)
{ ... }
```

- Possible optimisation: *conditional chunks*
  - Single path with chunk $tc.x \mapsto tx \# (tb\ ?100 : 0)$
  - Disadvantage: Now every field dereferencing requires a Z3 invocation in order to check if we have non-zero permissions
  - Prototypical implementation looked promising
    - verification time of massively branching programs dropped significantly
    - verification time of other programs increased slightly

Tools

- Latest Chalice version uses a new permission model not yet supported by Syxc, hence we had to use a slightly outdated Chalice version
- Syxc uses Chalice (as a library) to parse input into an AST
- Recent Boogie version; limited to one error per Chalice method
- Z3 3.1, smtlib2 frontend via std-io, interaction is logged in a file
- Syxc uses nearly the same Z3 settings as Chalice does, except
  - Syxc requires Z3 to respond to every command, not only to `check-sat`
  - Syxc uses global declarations, not scope-local ones
- Other differences:
  - Syxc encodes snapshots, references and lists as Z3 integers
    → might increase the number of quantifier instantiations
  - Syxc uses Boogie's sequence axiomatisation, but they range over integers only, whereas Boogie's are polymorphic
    → might increase the workload for Z3

Results

1st block: tests expected to fail

2nd block: tests expected to succeed

File			Syxc			Chalice			Syxc rel. to Chalice		
Name	LOC	Meth.	Time (s)	QI	Conflicts	Time (s)	QI	Conflicts	Time (%)	QI (%)	Conflicts (%)
Cell	30	3	1.12	13	1	2.39	410	21	46.86	3.17	4.76
LoopLockChange	[illegible]	3	[illegible].12	134	9	2.47	1519	90	45.34	8.82	10.00
ProdConsChannel	60	6	1.24	47	10	2.51	1009	72	49.40	4.66	13.89
Prog1	53	7	1.12	28	6	2.41	1045	97	46.47	2.68	6.19
Prog2	26	4	1.04	7	3	2.25	124	12	46.22	5.65	25.00
Prog3	20	3	0.98	8	1	2.25	128	7	43.56	6.25	14.29
Prog4	36	3	1.09	25	4	2.35	375	47	46.38	6.67	8.51
AVLTree.iterative	212	3	2.24	231	177	9.45	146238	1288	23.70	0.16	13.72
AVLTree.nokeys	572	19	34.21	2357	4304	154.03	2541875	19593	22.21	0.09	21.97
Cell	131	[illegible]	1.78	233	40	3.23	9599	526	55.11	2.43	7.60
CopyLessMessagePassing	54	3	1.46	218	35	2.72	4769	210	53.68	4.57	16.67
CopyLessMessagePassing-w/a	62	3	1.71	751	75	2.76	4014	181	61.96	18.71	41.44
CopyLessMessagePassing-w/a2	66	3	1.66	792	78	2.92	10542	325	56.85	7.51	24.00
Dining-philosophers	74	3	1.62	1181	128	3.09	17144	416	52.43	6.89	30.82
Iterator	123	6	3.65	436	210	3.27	7577	316	111.62	5.75	66.39
Iterator2	111	6	1.69	174	41	3.17	12867	304	53.31	1.35	13.49
LoopLockChange	69	5	1.26	220	42	2.60	2624	207	48.46	8.38	20.29
OwickiGries	31	2	1.21	121	41	2.61	5696	206	46.36	2.12	19.90
PetersonsAlgorithm	70	3	2.12	1196	278	7.36	160574	1500	28.80	0.74	18.53
ProdConsChannel	78	6	1.43	282	117	2.78	4640	316	51.44	6.08	37.03
Producer-consumer	171	11	1.91	466	147	3.84	17667	598	49.74	2.64	24.58
Quantifiers	28	1	1.02	16	4	2.23	196	12	45.74	8.16	33.33
RockBand	109	13	1.30	79	14	2.92	9129	341	44.52	0.87	4.11
Sieve	56	4	1.42	308	93	2.74	7596	246	51.82	4.05	37.80
Swap	19	2	0.98	10	2	2.20	261	20	44.55	3.83	10.00
Prog1	33	4	1.07	40	27	2.32	1101	106	46.12	3.63	25.47
Prog2	52	7	1.11	22	17	2.31	258	28	48.05	8.53	60.71
Prog3	163	16	1.58	627	144	3.53	21632	640	44.76	2.90	22.50
Prog4	19	1	1.07	76	22	2.32	877	61	46.12	8.67	36.07