SMT-based Model Checking of Transition Systems

Timothy.Bourke@inria.fr

3 October 2023

Specifying Properties

SMT Solver Basics

Model Checking

Bounded Model Checking and k-induction

Model Checking Lustre Programs: Kind 2

Two types of properties

Safety property: “Something bad never happens”

I.e., a property is invariant and true in any accessible state. E.g.:

- “The variable *temp* is always less than 101.”
- “The variable *temp* never increases by more than 5 in a single step.”

Liveness property: “Something good eventually happens.”

I.e., every execution will reach a state where the property holds.

- “If *heat* is on, *temp* eventually exceeds 10.”

Remark:

“If *heat* is on, *temp* exceeds 10 within 5 minutes.” is a safety property.

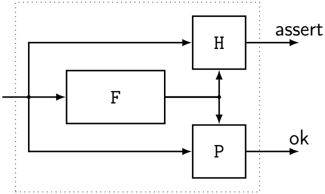
And remember that liveness properties are likely to be the least important part of your specification. You will probably not lose much if you simply omit them.

[Lamport (2002): Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers]

Synchronous Observers

- if $y = F(x)$, we write $ok = P(x,y)$ for the property relating x and y
- and $\texttt{assert}(H(x,y))$ to states an hypothesis on the environment.

```
node check(x:t) returns (ok:bool);
let
  assert H(x,y);
  y = F(x);
  ok = P(x,y);
tel;
```



If *assert* remains indefinitely true then *ok* remains indefinitely true $\texttt{always}(assert) \Rightarrow \texttt{always}(ok)$.

Any safety property can be expressed as a Lustre program. No need to introduce a temporal logic in the language

[Halbwachs, Lagnier, and Raymond (1993): Synchronous observers and the verification of reactive systems] [Halbwachs, Lagnier, and Ratel (1992): Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE];

Temporal properties are regular Lustre programs

Specifying Properties

SMT Solver Basics

Model Checking

Bounded Model Checking and *k*-induction

Model Checking Lustre Programs: Kind 2

SAT solvers

Given a boolean formula b with free variables $x_1, ..., x_n$ from propositional logic, find a valuation $V : \{x_1, ..., x_n\} \rightarrow \{0, 1\}$ such that $V(b) = 1$.

- initial algorithm by Davis-Putnam-Logemann-Loveland (DPLL); various heuristics. Generalization of SAT to QBF (Quantified Boolean Formula)
- a very active/competitive research/industrial topic (see http://www.satlive.org/)
- Now, more interest for SMT (Satisfiability Modulo Theory) for first-order logic (quantified formula + interpreted/non-interpreted functions)
- Close interaction between a SAT solver and ad-hoc solvers (e.g., simplex. method for linear arithmetic constraints)

SMT: Satisfiability Modulo Theories

- SAT = Satisfiability (of Boolean formulas)
- SMT = SAT Modulo Theories
- Input: set of constraints (interpreted in a theory)
- Output: are the constraints satisfiable?
  - `sat` and a *model* (an assignment to free variables that satisfies the constraints)
  - `unsat`: no model exists
  - `unknown`: could not determine due to resource limits, incompleteness, etcetera.
- Different solvers:
  - z3 (see also: docs and version in browser)
  - Alt-Ergo
  - CVC5
  - Yices
- Today we will use Z3 and SMT-LIB.

SMT-LIB 2.6

- SMT-LIB defines a common language for interfacing with SMT solvers

  [Barrett, Fontaine, and Tinelli (2021): The SMT-LIB Standard: Version 2.6] https://smtlib.cs.uiowa.edu/
- Developed to facilitate research and development in SMT (in particular, by providing an extensive benchmarking library)
- Lisp-like syntax for
  - a many-sorted first-order logic with equality
  - solver commands
  - declaring theory interfaces
- Solvers like Z3 also provide programmatic interfaces (e.g., Python, OCaml)

Satisfiability: true for some assignment

A `.smt2` file is a sequence of commands. (Fig. 3.6, p. 45 [Barrett, Fontaine, and Tinelli (2021): The SMT-LIB Standard: Version 2.6])

```
(declare-fun a () Bool)    ; uninterpreted function with zero arguments
(declare-const b Bool)     ; similar effect, easier to read

(assert (or a b))
(assert (= a false))

(echo "Is (a or b) and (a = false) satisfiable?")
(check-sat)
(get-model)
```

Try `z3 a_or_b.smt2`...

z3 looks for a model (an interpretation of the functions) that satisfies all the constraints.

Validity: true for all assignments

What about proving one of De Morgan's laws? $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$

```
(declare-const P Bool)
(declare-const Q Bool)

(assert (= (not (or P Q)) (and (not P) (not Q))))
(check-sat)
```

z3 says `sat`. Have we proved the law?

Validity: true for all assignments

What about proving one of De Morgan's laws? $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$

```
(declare-const P Bool)
(declare-const Q Bool)

(assert (= (not (or P Q)) (and (not P) (not Q))))
(check-sat)
```

z3 says `sat`. Have we proved the law?

```
(declare-const P Bool)
(declare-const Q Bool)

(assert (not (= (not (or P Q)) (and (not P) (not Q)))))
(check-sat)
```

Now z3 says `unsat`. Have we proved the law?

Validity: true for all assignments

What about proving one of De Morgan's laws? $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$

```
(declare-const P Bool)
(declare-const Q Bool)

(assert (= (not (or P Q)) (and (not P) (not Q))))
(check-sat)
```

z3 says `sat`. Have we proved the law?

```
(declare-const P Bool)
(declare-const Q Bool)

(assert (not (= (not (or P Q)) (and (not P) (not Q)))))
(check-sat)
```

Now z3 says `unsat`. Have we proved the law?

Yes. There are no values for P and Q such that the law is not true.

Satisfiability and Validity

$$\begin{aligned}
\text{satisfiable}(b) &\overset{\text{def}}{=} \exists V, V(b) = 1 \\
\text{valid}(b) &\overset{\text{def}}{=} \forall V, V(b) = 1
\end{aligned}$$

$$\begin{aligned}
\text{valid}(b) &= \neg\neg\left(\forall V, V(b) = 1\right) \\
&= \neg\left(\exists V, \neg\left(V(b) = 1\right)\right) \\
&= \neg\,\text{satisfiable}(\neg b)
\end{aligned}$$

Satisfiability and Validity

$$\begin{aligned} \text{satisfiable}(b) &\overset{\text{def}}{=} \exists V, V(b) = 1 \\ \text{valid}(b) &\overset{\text{def}}{=} \forall V, V(b) = 1 \end{aligned}$$

$$\begin{aligned} \text{valid}(b) &= \neg\neg\,(\forall V, V(b) = 1) \\ &= \neg\,(\exists V, \neg\,(V(b) = 1)) \\ &= \neg\,\text{satisfiable}(\neg b) \end{aligned}$$

$$\begin{aligned} \text{satisfiable}(b) &= \neg\neg\,(\exists V, V(b) = 1) \\ &= \neg\,(\forall V, \neg\,(V(b) = 1)) \\ &= \neg\,\text{valid}(\neg b) \end{aligned}$$

Satisfiability and Validity

$$\begin{aligned}\text{satisfiable}(b) &\overset{\text{def}}{=} \exists V, V(b) = 1 \\ \text{valid}(b) &\overset{\text{def}}{=} \forall V, V(b) = 1\end{aligned}$$

$$\begin{aligned}\text{valid}(b) &= \neg\neg(\forall V, V(b) = 1) \\ &= \neg(\exists V, \neg(V(b) = 1)) \\ &= \neg\,\text{satisfiable}(\neg b)\end{aligned} \qquad \begin{aligned}\text{satisfiable}(b) &= \neg\neg(\exists V, V(b) = 1) \\ &= \neg(\forall V, \neg(V(b) = 1)) \\ &= \neg\,\text{valid}(\neg b)\end{aligned}$$

To determine $\text{valid}(P \wedge Q \Rightarrow R)$, ask $\text{satisfiable}(P \wedge Q \wedge \neg R)$ and require `unsat`.

Satisfiability and Validity

$$\text{satisfiable}(b) \overset{\text{def}}{=} \exists V, V(b) = 1$$
$$\text{valid}(b) \overset{\text{def}}{=} \forall V, V(b) = 1$$

$$\begin{aligned}\text{valid}(b) &= \neg\neg(\forall V, V(b) = 1)\\ &= \neg(\exists V, \neg(V(b) = 1))\\ &= \neg\,\text{satisfiable}(\neg b)\end{aligned}$$

$$\begin{aligned}\text{satisfiable}(b) &= \neg\neg(\exists V, V(b) = 1)\\ &= \neg(\forall V, \neg(V(b) = 1))\\ &= \neg\,\text{valid}(\neg b)\end{aligned}$$

To determine $\text{valid}(P \wedge Q \Rightarrow R)$, ask $\text{satisfiable}(P \wedge Q \wedge \neg R)$ and require `unsat`.

$$(A \Rightarrow B \overset{\text{def}}{=} \neg A \vee B)$$
$$\begin{aligned}\text{valid}(P \wedge Q \Rightarrow R) &= \neg\,\text{satisfiable}(\neg(\neg(P \wedge Q) \vee R))\\ &= \neg\,\text{satisfiable}((P \wedge Q) \wedge \neg R)\end{aligned}$$

If `sat`, try `(get-model)`. Can also use `(check-sat-assuming ((and P Q) R))`.

Interacting with the solver

- Typical to run several `(check-sat)` commands in series.
- Use `(push)` and `(pop)` to manage the environment of functions and assertions.

```
(declare-const P Bool)
(declare-const Q Bool)

(push)
(assert (not (= (not (or P Q)) (and (not P) (not Q)))))
(echo "Checking: !(P or Q) <=> !P and !Q (unsat = valid)")
(check-sat)
(pop)

(push)
(assert (not (= (not (and P Q)) (or (not P) (not Q)))))
(echo "Checking: !(P and Q) <=> !P or !Q (unsat = valid)")
(check-sat)
(pop)
```

- Usually interact with the solver using a programmatic interface.
  Query results determine future queries.
- Solvers are designed to work incrementally.

Functions

- Functions declared with `declare-fun` are uninterpreted.
- Functions from theories, like `xor`, are interpreted.

  see https://smtlib.cs.uiowa.edu/theories-Core.shtml

```
(declare-fun f (Bool Bool) Bool)
(assert (and (= (f false false) false)
             (= (f false true)  true)
             (= (f true  false) true)
             (= (f true  true)  false)))

(declare-const a Bool)
(declare-const b Bool)
(assert (not (= (f a b) (xor a b))))
(check-sat)
```

Functions

- Functions declared with `declare-fun` are uninterpreted.
- Functions from theories, like `xor`, are interpreted.

  see https://smtlib.cs.uiowa.edu/theories-Core.shtml

```
(declare-fun f (Bool Bool) Bool)
(assert (and (= (f false false) false)
             (= (f false true)  true)
             (= (f true  false) true)
             (= (f true  true)  false)))

(declare-const a Bool)
(declare-const b Bool)
(assert (not (= (f a b) (xor a b))))
(check-sat)
```

- Can also define functions:

```
(define-fun f ((x Bool) (y Bool)) Bool (xor x y))
```

Terms and Formulas

```
⟨qual_identifier⟩  ::=  ⟨identifier⟩  |  ( as ⟨identifier⟩ ⟨sort⟩ )
⟨var_binding⟩      ::=  ( ⟨symbol⟩ ⟨term⟩ )
⟨sorted_var⟩       ::=  ( ⟨symbol⟩ ⟨sort⟩ )
⟨pattern⟩          ::=  ⟨symbol⟩  |  ( ⟨symbol⟩ ⟨symbol⟩+ )
⟨match_case⟩       ::=  ( ⟨pattern⟩ ⟨term⟩ )
⟨term⟩             ::=  ⟨spec_constant⟩
                     |  ⟨qual_identifier⟩
                     |  ( ⟨qual_identifier⟩ ⟨term⟩+ )
                     |  ( let ( ⟨var_binding⟩+ ) ⟨term⟩ )
                     |  ( forall ( ⟨sorted_var⟩+ ) ⟨term⟩ )
                     |  ( exists ( ⟨sorted_var⟩+ ) ⟨term⟩ )
                     |  ( match ⟨term⟩ ( ⟨match_case⟩+ ) )
                     |  ( ! ⟨term⟩ ⟨attribute⟩+ )
```

(p. 27, [Barrett, Fontaine, and Tinelli (2021): The SMT-LIB Standard: Version 2.6])

- Satisfiability without quantifiers is NP-Complete
- With quantifiers it is undecidable.
- The effectiveness of *quantifier elimination* depends on the shape of formulas.
- Take care with your encodings!

Exercise: model checking 1-bit adders

How to be sure that `full_add` and `full_add_h` are equivalent?

$$\forall a, b, c : \texttt{bool}.\, \texttt{full\_add}(a, b, c) = \texttt{full\_add\_h}(a, b, c)$$

Implement the following interface so that it returns `true` exactly when two full adder implementations return the same value for the same inputs.

```
-- file fulladder.lus
node equivalence(a,b,c:bool) returns (ok:bool);
  var o1, c1, o2, c2: bool;
  let
    (o1, c1) = full_add(a,b,c);
    (o2, c2) = full_add_h(a,b,c);
    ok = (o1 = o2) and (c1 = c2);
  tel;
```

Check equivalence with z3 and SMT-LIB!

Specifying Properties

SMT Solver Basics

Model Checking

Bounded Model Checking and k-induction

Model Checking Lustre Programs: Kind 2

Model Checking: (extremely) partial overview

1981 Explicit state enumeration

[E. M. Clarke and Emerson (1981): Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic] [Queille and Sifakis (1982): Specification and Verification of Concurrent Systems in CESAR]

1992 BDD-based algorithms

[Burch, E. Clarke, McMillan, Dill, and Hwang (1992): Symbolic Model Checking: 10^{20} States and Beyond]

1999 Bounded Model Checking

[Biere, Cimatti, E. Clarke, and Zhu (1999): Symbolic Model Checking without BDDs]

2000 K-induction

[Sheeran, Singh, and Stålmarck (2000): Checking Safety Properties Using Induction and a SAT-Solver]

2003 Interpolation-based

[McMillan (2003): Interpolation and SAT-based model checking]

2011 IC3 Algorithm

[Bradley (2011): SAT-Based Model Checking without Unrolling]

Model checking of Lustre

- Lesar: based on BDDs

  [Halbwachs, Lagnier, and Ratel (1992): Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE]

- Kind 2: based on SMT/k-induction/IC3

  [Champion, Mebsout, Sticksel, and Tinelli (2016): The Kind 2 Model Checker]

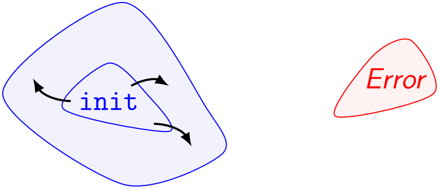
- DV of (Ansys) Scade based on Prover SAT/k-induction

Model checking: forward method

The set of reachable states never intersects the set of error states

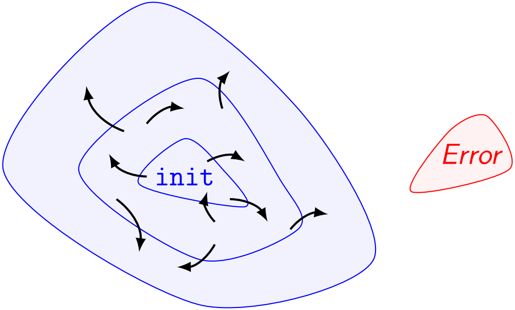
Model checking: forward method

The set of reachable states never intersects the set of error states



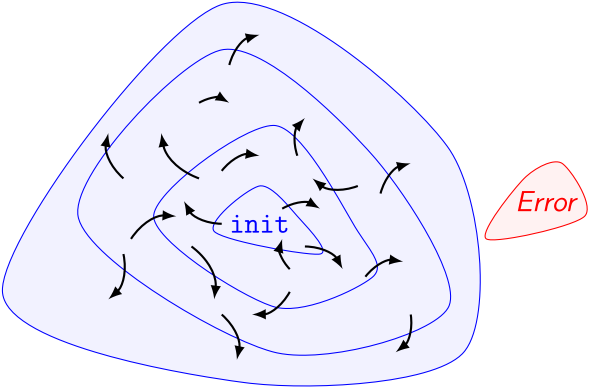
Model checking: forward method

The set of reachable states never intersects the set of error states



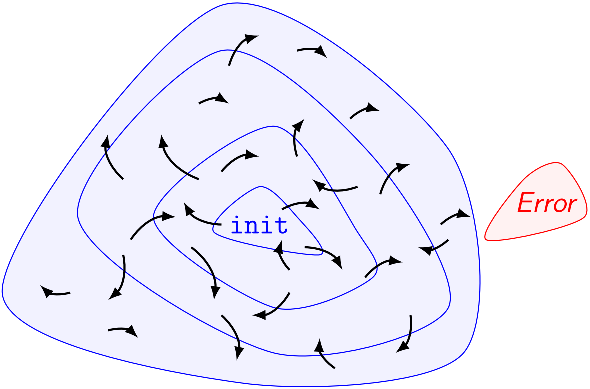
Model checking: forward method

The set of reachable states never intersects the set of error states



Model checking: forward method

The set of reachable states never intersects the set of error states

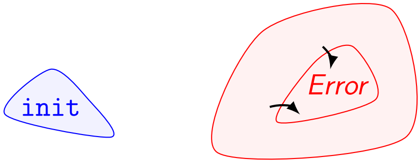


Model checking: backward method

The states that can reach an error state do not include the initial states

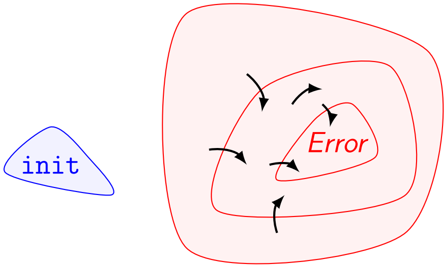
Model checking: backward method

The states that can reach an error state do not include the initial states



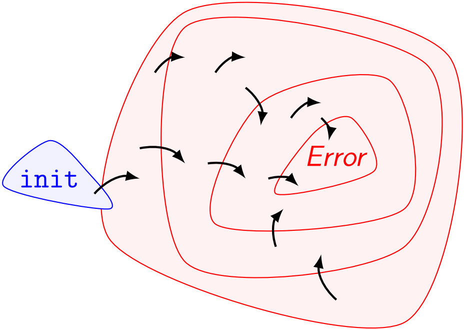
Model checking: backward method

The states that can reach an error state do not include the initial states

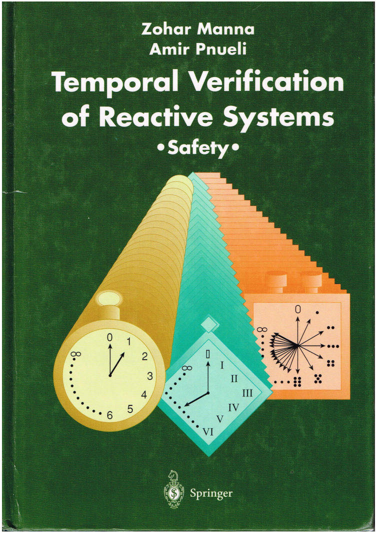


Model checking: backward method

The states that can reach an error state do not include the initial states

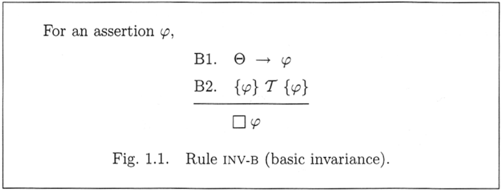


Verifying safety properties of reactive systems



- Published in 1995
  [Manna and Pnueli (1995): Temporal Verification of Reactive Systems: Safety]
- Companion to
  [Manna and Pnueli (1992): The Temporal Logic of Reactive and Concurrent Systems]
- Builds on Floyd’s inductive invariants
- Temporal logic formulas as ‘proof patterns’

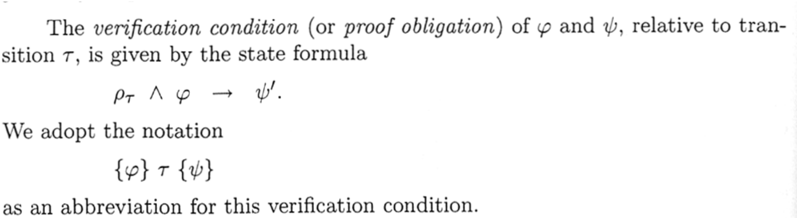
The basic 'pattern' for showing invariance



For an assertion φ,

$$\begin{array}{ll} \text{B1.} & \Theta \rightarrow \varphi \\ \text{B2.} & \{\varphi\}\ \mathcal{T}\ \{\varphi\} \\ \hline & \Box\varphi \end{array}$$

Fig. 1.1. Rule INV-B (basic invariance).



The *verification condition* (or *proof obligation*) of φ and ψ, relative to transition τ, is given by the state formula

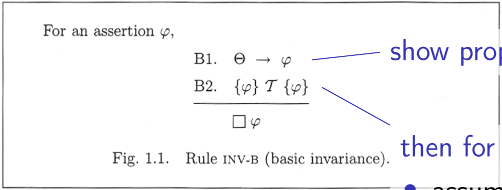
$$\rho_\tau \wedge \varphi \rightarrow \psi'.$$

We adopt the notation

$$\{\varphi\}\ \tau\ \{\psi\}$$

as an abbreviation for this verification condition.

The basic 'pattern' for showing invariance



For an assertion φ,

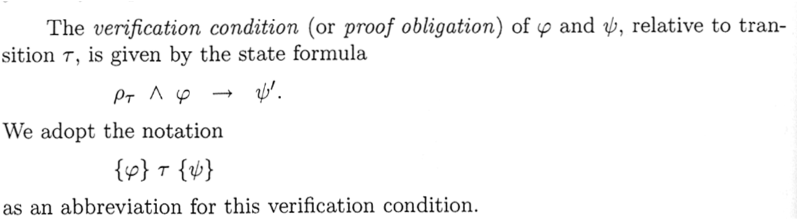
$$\begin{array}{ll} \text{B1.} & \Theta \rightarrow \varphi \\ \text{B2.} & \{\varphi\}\ \mathcal{T}\ \{\varphi\} \\ \hline & \Box\,\varphi \end{array}$$

Fig. 1.1. Rule INV-B (basic invariance).

show property of initial states

then for every transition:

- assume the property of the pre state (φ)
- show the property of the post state (φ')



The *verification condition* (or *proof obligation*) of φ and ψ, relative to transition τ, is given by the state formula

$$\rho_\tau \wedge \varphi \rightarrow \psi'.$$

We adopt the notation

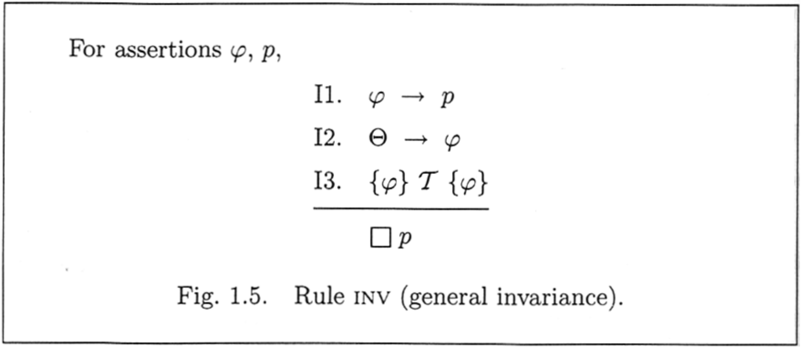
$$\{\varphi\}\ \tau\ \{\psi\}$$

as an abbreviation for this verification condition.

Exercise: proving invariance of a simple transition system

- Consider a simple transition system with two integer state variables x and y:
  $\mathit{init}(x, y) := (x = 1) \land (y = 1)$
  $\mathit{trans}(x, y, x', y') := (x' = x + 1) \land (y' = y + x)$
- And the safety property $\mathit{prop}(x, y) = y \geq 1$.
- Encode this system and use Z3 to prove that the property is invariant.

General rule for showing invariance



For assertions φ, p,

$$\begin{array}{ll} \text{I1.} & \varphi \rightarrow p \\ \text{I2.} & \Theta \rightarrow \varphi \\ \text{I3.} & \{\varphi\}\ \mathcal{T}\ \{\varphi\} \\ \hline & \Box\, p \end{array}$$

Fig. 1.5. Rule INV (general invariance).

Not all *invariants* are *inductive invariants*.

Inductive invariants and model checking

- This idea works for manual/interactive proof.
- What about automatic proof (model checking)?
- (BTW, note that SMT solvers do not themselves do induction.)

- k-induction: strengthen P with information from last k steps.
  [Sheeran, Singh, and Stålmarck (2000): Checking Safety Properties Using Induction and a SAT-Solver]
- IC3: automate 'discovery' of strengthenings
  [Bradley (2011): SAT-Based Model Checking without Unrolling]

- Generic algorithms
  - work with SAT solvers on boolean transition systems, or
  - with SMT solvers on richer transition systems.
  - avoid or minimize quantifiers, look for efficient encodings

Specifying Properties

SMT Solver Basics

Model Checking

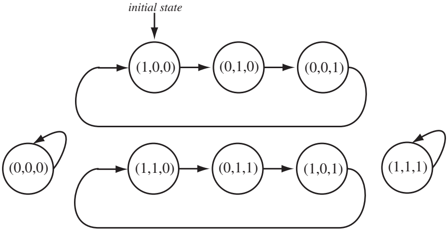
Bounded Model Checking and k-induction

Model Checking Lustre Programs: Kind 2

k-induction

- Iterate BMC. Explained as a succession of algorithms.
  [Sheeran, Singh, and Stålmarck (2000): Checking Safety Properties Using Induction and a SAT-Solver]
- Focus completely on invariant properties (AG f)

```
node ring_counter()
returns (a, b, c : bool);
let
    a = true fby c;
    b = false fby a;
    c = false fby b;
tel
```



k-induction: Algorithm 1

Algorithm 1 First algorithm to check if system is P-safe

```
$i$=0
while True do
    if not Sat($I(s_0) \land loopFree(s_{[0..i]})$) or not Sat(($loopFree(s_{[0..i]}) \land \neg P(s_i)$)) then
        return True
    end if
    if Sat($I(s_0) \land path(s_{[0..i]}) \land \neg P(s_i)$) then
        return Trace $c_{[0..i]}$
    end if
    $i = i + 1$
end while
```

$$path(s_{[0..n]}) \hat{=} \bigwedge_{0 \leq i < n} T(s_i, s_{i+1})$$

$$loopFree(s_{[0..n]}) \hat{=} path(s_{[0..n]}) \land \bigwedge_{0 \leq i < j \leq n} s_i \neq s_j$$

The restriction to loop-free paths is necessary for completeness.

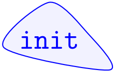
- Check for existence of loop-free path.
- Check for existence of bad path.

k-induction: Algorithm 1

Algorithm 1 First algorithm to check if system is P-safe

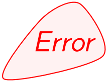
```
$i$=0
**while** True **do**
    **if** not Sat($I(s_0) \wedge loopFree(s_{[0..i]})$) or not Sat(($loopFree(s_{[0..i]}) \wedge \neg P(s_i)$)) **then**
        return True
    **end if**
    **if** Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg P(s_i)$) **then**
        return Trace $c_{[0..i]}$
    **end if**
    $i = i + 1$
**end while**
```

$$path(s_{[0..n]}) \ \hat{=} \ \bigwedge_{0 \leq i < n} T(s_i, s_{i+1})$$



$$loopFree(s_{[0..n]}) \ \hat{=} \ path(s_{[0..n]}) \ \wedge \bigwedge_{0 \leq i < j \leq n} s_i \neq s_j$$

The restriction to loop-free paths is necessary for completeness.



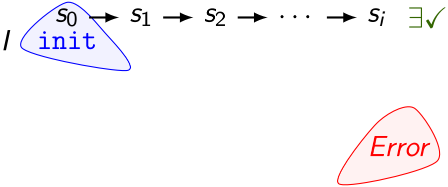
k-induction: Algorithm 1

Algorithm 1 First algorithm to check if system is P-safe

```
$i$=0
while True do
    if not Sat($I(s_0) \land \mathit{loopFree}(s_{[0..i]})$) or not Sat(($\mathit{loopFree}(s_{[0..i]}) \land \neg P(s_i)$)) then
        return True
    end if
    if Sat($I(s_0) \land \mathit{path}(s_{[0..i]}) \land \neg P(s_i)$) then
        return Trace $c_{[0..i]}$
    end if
    $i = i + 1$
end while
```

$$\mathit{path}(s_{[0..n]}) \ \hat{=} \bigwedge_{0 \le i < n} T(s_i, s_{i+1})$$

$$\mathit{loopFree}(s_{[0..n]}) \ \hat{=} \ \mathit{path}(s_{[0..n]}) \land \bigwedge_{0 \le i < j \le n} s_i \neq s_j$$



The restriction to loop-free paths is necessary for completeness.

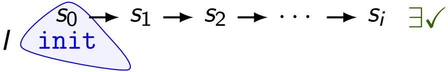
k-induction: Algorithm 1

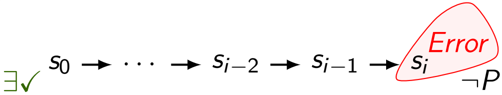
Algorithm 1 First algorithm to check if system is P-safe

```
$i$=0
while True do
    if not Sat($I(s_0) \wedge loopFree(s_{[0..i]})$) or not Sat(($loopFree(s_{[0..i]}) \wedge \neg P(s_i)$)) then
        return True
    end if
    if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg P(s_i)$) then
        return Trace $c_{[0..i]}$
    end if
    $i = i + 1$
end while
```

$$path(s_{[0..n]}) \ \hat{=} \bigwedge_{0 \le i < n} T(s_i, s_{i+1})$$

$$loopFree(s_{[0..n]}) \ \hat{=} \ path(s_{[0..n]}) \ \wedge \bigwedge_{0 \le i < j \le n} s_i \neq s_j$$





The restriction to loop-free paths is necessary for completeness.

k-induction: Algorithm 1

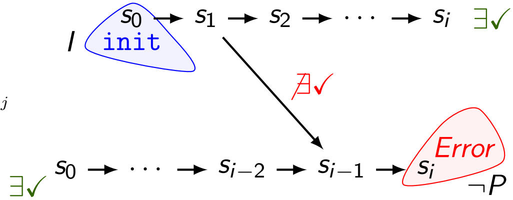
Algorithm 1 First algorithm to check if system is P-safe

```
$i$=0
while True do
    if not Sat($I(s_0) \wedge loopFree(s_{[0..i]})$) or not Sat(($loopFree(s_{[0..i]}) \wedge \neg P(s_i)$)) then
        return True
    end if
    if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg P(s_i)$) then
        return Trace $c_{[0..i]}$
    end if
    $i = i + 1$
end while
```

$$path(s_{[0..n]}) \ \hat{=} \ \bigwedge_{0 \le i < n} T(s_i, s_{i+1})$$

$$loopFree(s_{[0..n]}) \ \hat{=} \ path(s_{[0..n]}) \ \wedge \bigwedge_{0 \le i < j \le n} s_i \neq s_j$$

The restriction to loop-free paths is necessary for completeness.



k-induction: Algorithm 2

Algorithm 1 First algorithm to check if system is P-safe

```
$i$=0
while True do
  if not Sat($I(s_0) \wedge loopFree(s_{[0..i]})$) or not Sat(($loopFree(s_{[0..i]}) \wedge \neg P(s_i)$) then
    return True
  end if
  if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg P(s_i)$) then
    return Trace $c_{[0..i]}$
  end if
  $i = i + 1$
end while
```

Algorithm 2 An improved algorithm to check if system is P-safe

```
$i$=0
while True do
  if not Sat($I(s_0) \wedge all.\neg I(s_{[1..i]}) \wedge loopFree(s_{[0..i]})$)
  or not Sat(($loopFree(s_{[0..i]}) \wedge all.P(s_{[0..(i-1)]}) \wedge \neg P(s_i)$) then
    return True
  end if
  if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg P(s_i)$) then
    return Trace $c_{[0..i]}$
  end if
  $i = i + 1$
end while
```

$$path(s_{[0..n]}) \doteq \bigwedge_{0 \le i < n} T(s_i, s_{i+1})$$

$$loopFree(s_{[0..n]}) \doteq path(s_{[0..n]}) \wedge \bigwedge_{0 \le i < j \le n} s_i \neq s_j$$

- Exclude forward paths that loop back through initial states.
- Exclude backward paths that loop back through error states.
- I.e., tighten the termination conditions.

k-induction: Algorithm 3

Algorithm 2 An improved algorithm to check if system is P-safe

```
$i$=0
while True do
  if not Sat($I(s_0) \wedge all.\neg I(s_{[1..i]}) \wedge loopFree(s_{[0..i]})$)
  or not Sat(($loopFree(s_{[0..i]}) \wedge all.P(s_{[0..(i-1)]}) \wedge \neg P(s_i)$) then
    return True
  end if
  if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg P(s_i)$) then
    return Trace $c_{[0..i]}$
  end if
  $i = i + 1$
end while
```

Algorithm 3 An algorithm that need not iterate from 0

```
$i$= some constant which can be greater than zero
while True do
  if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg all.P(s_{[0..i]})$) then
    return Trace $c_{[0..i]}$
  end if
  if not Sat($I(s_0) \wedge all.\neg I(s_{[1..(i+1)]}) \wedge loopFree(s_{[0..(i+1)]})$)
  or not Sat(($loopFree(s_{[0..(i+1)]}) \wedge all.P(s_{[0..i]}) \wedge \neg P(s_{i+1})$) then
    return True
  end if
  $i = i + 1$
end while
```

$$path(s_{[0..n]}) \;\hat{=}\; \bigwedge_{0 \le i < n} T(s_i, s_{i+1})$$

$$loopFree(s_{[0..n]}) \;\hat{=}\; path(s_{[0..n]}) \;\wedge \bigwedge_{0 \le i < j \le n} s_i \neq s_j$$

- Start an any i
- Swap order of checks (`ifs`)
- Check proposition along entire path: $\forall_{0 \le j \le i}, P(s_j)$
- Extend loop-free check to $i + 1$

k-induction: Algorithm 4

Algorithm 3 An algorithm that need not iterate from 0

```
$i$= some constant which can be greater than zero
while True do
    if Sat($I(s_0) \wedge path(s_{[0..i]}) \wedge \neg all.P(s_{[0..i]})$) then
        return Trace $c_{[0..i]}$
    end if
    if not Sat($I(s_0) \wedge all.\neg I(s_{[1..(i+1)]}) \wedge loopFree(s_{[0..(i+1)]})$)
    or not Sat($(loopFree(s_{[0..(i+1)]}) \wedge all.P(s_{[0..i]}) \wedge \neg P(s_{i+1})$) then
        return True
    end if
    $i = i + 1$
end while
```

$$path(s_{[0..n]}) \ \hat{=} \bigwedge_{0 \leq i < n} T(s_i, s_{i+1})$$

$$loopFree(s_{[0..n]}) \ \hat{=} \ path(s_{[0..n]}) \ \wedge \bigwedge_{0 \leq i < j \leq n} s_i \neq s_j$$

Algorithm 4 A forwards version of the algorithm

```
$i$= some constant which can be greater than zero
while True do
    if Sat($\neg(I(s_0) \wedge path(s_{[0..i]}) \rightarrow all.P(s_{[0..i]}))$) then
        return Trace $c_{[0..i]}$
    end if
    if Taut($\neg I(s_0) \leftarrow all.\neg I(s_{[1..(i+1)]}) \wedge loopFree(s_{[0..(i+1)]})$)
    or Taut($(loopFree(s_{[0..(i+1)]}) \wedge all.P(s_{[0..i]}) \rightarrow P(s_{i+1})$) then
        return True
    end if
    $i = i + 1$
end while
```

- Reformulate checks as implications
- The first check is the base case of the induction.
- The second is the transition case, and also a check that a loop-free path of length *i* exists.

k-induction and completeness

- The algorithm is complete for finite transition systems.
- Diameter = length of the longest shortest path in transition system.

$$shortest(s_{[0..n]}) \;\hat{=}\; path(s_{[0..n]}) \wedge \neg(\bigvee_{0 \le i < n} path_i(s_0, s_n))$$

- Two extra algorithms that only consider shortest paths, but they require quantifier elimination.

Specifying Properties

SMT Solver Basics

Model Checking

Bounded Model Checking and *k*-induction

Model Checking Lustre Programs: Kind 2

Model checking Lustre programs: Kind 2

- http://kind2-mc.github.io/kind2/ (or use web interface: http://kind.cs.uiowa.edu:8080/app/)
- SMT-based Model Checker for Lustre: BMC, k-induction, IC3, . . .
- Specify properties to check as comments:

  ——%PROPERTY ok;

```
> kind2 toggles.lus
kind2 v1.1.0-214-g00b3d21d

===============================================================
Analyzing compare
  with First top: "compare"
             subsystems
               | concrete: toggle2, toggle1

<Success> Property ok is valid by inductive step after 0.164s.

---------------------------------------------------------------
Summary of properties:
---------------------------------------------------------------
ok: valid (at 1)
===============================================================
> kind2 --enable BMC --enable IND --lus_main compare toggles.lus
```

Kind 2

- Consider integers (not machine words)
- and infinite-precision rationals (not floating-point)

- Optimize existing techniques for Lustre programs and features of modern SMT solvers.

Encoding Lustre in SMT [Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]

- Represent streams as uninterpreted functions $\mathbb{N} \to \tau$
- Examples:

  `x = y + z` $\forall n : \mathbb{N},\ x(n) = y(n) + z(n)$

  `x = y -> y + pre z` $\forall n : \mathbb{N},\ x(n) = ite(n = 0, y(0), y(n) + z(n - 1))$

Encoding Lustre in SMT [Hagen and Tinelli (2008): Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques]

- Represent streams as uninterpreted functions $\mathbb{N} \to \tau$
- Examples:

  x = y + z $\quad \forall n : \mathbb{N},\ x(n) = y(n) + z(n)$

  x = y -> y + pre z $\quad \forall n : \mathbb{N},\ x(n) = ite(n = 0, y(0), y(n) + z(n-1))$

- Let N be a node with stream variables $\mathrm{x} = \langle x_1, \ldots, x_p, y_1, \ldots, y_q \rangle$ ($x_1, \ldots, x_p$ are inputs, and $y_1, \ldots, y_q$ are outputs)
- $$\Delta(n) = \begin{cases} y_1(n) = t_1[\mathrm{x}(n), \mathrm{x}(n-1), \ldots, \mathrm{x}(n-d)] \\ \qquad\vdots \\ y_q(n) = t_q[\mathrm{x}(n), \mathrm{x}(n-1), \ldots, \mathrm{x}(n-d)] \end{cases}$$

```
node thermostat (actual_temp, target_temp, margin: real)
returns (cool, heat: bool);
let
  cool = (actual_temp − target_temp) > margin;
  heat = (actual_temp − target_temp) < −margin;
tel

node therm_control (actual: real; up, down: bool) returns (heat, cool: bool);
var target, margin: real;
let
  margin = 1.5;
  target = 70.0 −> if down then (pre target) − 1.0
                         else if up then (pre target) + 1.0
                         else pre target;
  (cool, heat) = thermostat (actual, target, margin);
tel
```

$$\Delta(n) = \begin{cases} m(n) = 1.5 \\ t(n) = ite(n = 0, 70.0, ite(d(n), t(n-1) - 1.0, \ldots)) \\ c(n) = (a(n) - t(n)) > m(n) \\ h(n) = ((a(n) - t(n)) < -m(n) \end{cases}$$

SMT-based *k*-induction

$$\Delta_0 \land \Delta_1 \land \cdots \land \Delta_k \models_{\mathcal{IL}} P_0 \land P_1 \land \cdots \land P_k \tag{1}$$

$$\begin{array}{l} \Delta_\mathsf{n} \land \Delta_{\mathsf{n}+1} \land \cdots \land \Delta_{\mathsf{n}+(k+1)} \land \\ P_\mathsf{n} \land P_{\mathsf{n}+1} \land \cdots \land P_{\mathsf{n}+k} \end{array} \models_{\mathcal{IL}} P_{\mathsf{n}+(k+1)} \tag{2}$$

where $k \geq 0$ and *n* is an uninterpreted integer constant.

Kind 2 optimizations: path compression

$C_{n,k}$ is a predicate over state variables that is satisfied iff no two configurations in a path have the same state and none of them, except possibly the first is the initial state.

$$\begin{array}{l} \Delta_{\mathsf{n}} \wedge \Delta_{\mathsf{n}+1} \wedge \cdots \wedge \Delta_{\mathsf{n}+(k+1)} \wedge \\ P_{\mathsf{n}} \wedge P_{\mathsf{n}+1} \wedge \cdots \wedge P_{\mathsf{n}+k} \wedge C_{\mathsf{n},k} \end{array} \quad \models_{\mathcal{IL}} \quad P_{\mathsf{n}+(k+1)} \qquad (2\text{'})$$

Allows the addition of a termination condition.

$$\Delta_0 \wedge \cdots \wedge \Delta_k \quad \models_{\mathcal{IL}} \quad \neg C_{0,k+1}$$

Kind 2 optimizations: abstraction

- Drop equations defining variables that are not mentioned in the property P.
  Sound: those variables are unconstrained (like inputs).
- Add them back one-by-one if checking fails.
  Take one (removed) variable appearing in counter-example and recursively add removed variables from its defining expression (work towards input variables).

Summary

- Express programs, (safety) properties, and assumptions on the environment in a single language.
- Model-checking ideal:
  - 'push-button' verification gives ok or counter-example;
  - no need to understand why (i.e., write invariants).
- SAT-based techniques for BMC, complete with k-induction.
- Extend SAT to SMT to handle integers and directly encode Lustre programs.
- Lots of tools for automating induction and interfacing with SMT solvers
  - Mikino tutorial [Champion, Oliveira, and Didier (2022): Mikino: Induction for Dummies]
  - F* [Swamy et al. (2016): Dependent Types and Multi-monadic Effects in F*], Why3 [Bobot, Filliâtre, Marché, and Paskevich (2011): Why3: Sheperd your herd of provers], Boogie [Barnett, Chang, DeLine, Jacobs, and Leino (2005): Boogie: A Modular Reusable Verifier for Object-Oriented Programs], . . .
- Just the tip of the iceberg (IC3/PDR, interactive theorem provers, . . .)

References I

- Barnett, M., B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino (Nov. 2005). "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Proc. 4th Int. Symp. Formal Methods for Components and Objects (FMCO 2005)*. Vol. 4111. LNCS. Amsterdam, The Netherlands: Springer, pp. 364–387.
- Barrett, C., P. Fontaine, and C. Tinelli (May 2021). *The SMT-LIB Standard: Version 2.6*.
- Biere, A., A. Cimatti, E. Clarke, and Y. Zhu (Mar. 1999). "Symbolic Model Checking without BDDs". In: *5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1999)*. Ed. by W. R. Cleaveland. Vol. 1579. LNCS. Amsterdam, The Netherlands: Springer, pp. 193–207.
- Bobot, F., J.-C. Filliâtre, C. Marché, and A. Paskevich (Aug. 2011). "Why3: Sheperd your herd of provers". In: *Boogie 2011: First Int. Workshop on Intermediate Verification Languages*. Wrocław, Poland, pp. 53–64.
- Bradley, A. R. (Jan. 2011). "SAT-Based Model Checking without Unrolling". In: *Proc. 12th Int. Conf. on on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*. Ed. by R. Jhala and D. Schmidt. Vol. 6538. LNCS. Austin, TX, USA: Springer, pp. 70–87.

References II

- Burch, J., E. Clarke, K. McMillan, D. Dill, and J. Hwang (June 1992). “Symbolic Model Checking: 10^{20} States and Beyond”. In: *Information and Computation* 98.2, pp. 142–170.
- Champion, A., A. Mebsout, C. Sticksel, and C. Tinelli (July 2016). “The Kind 2 Model Checker”. In: *Proc. 28th Int. Conf. on Computer Aided Verification (CAV 2016), Part II*. Ed. by S. Chaudhuri and A. Farzan. Vol. 9780. LNCS. Toronto, Canada: Springer, pp. 510–517.
- Champion, A., S. de Oliveira, and K. Didier (June 2022). “Mikino: Induction for Dummies”. In: *$33^{ièmes}$ Journées Francophones des Langages Applicatifs (JFLA 2022)*. Ed. by C. Keller and T. Bourke. Saint-Médard-d’Excideuil, France, pp. 254–260.
- Clarke, E. M. and E. A. Emerson (May 1981). “Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic”. In: *Workshop on Logics of Programs*. Ed. by D. Kozen. Vol. 131. LNCS. Yorktown Heights, NY, USA: Springer, pp. 52–71.

References III

- Hagen, G. and C. Tinelli (Nov. 2008). "Scaling Up the Formal Verification of Lustre Programs with SMT-based Techniques". In: *Proc. 8th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2008)*. Ed. by A. Cimatti and R. B. Jones. IEEE. Portland, OR, USA, Article 15.
- Halbwachs, N., F. Lagnier, and P. Raymond (June 1993). "Synchronous observers and the verification of reactive systems". In: *Proc. 3rd Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*. Ed. by M. Nivat, C. Rattray, T. Rus, and G. Scollo. Twente: Workshops in Computing, Springer Verlag.
- Halbwachs, N., J.-C. Fernandez, and A. Bouajjani (Apr. 1993). "An executable temporal logic to express safety properties and its connection with the language Lustre". In: *Proc. 6th Int. Symp. Lucid and Intensional Programming (ISLIP'93)*. Quebec, Canada.
- Halbwachs, N., F. Lagnier, and C. Ratel (Sept. 1992). "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE". In: *IEEE Trans. Software Engineering* 18.9, pp. 785–793.

References IV

- Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley.
- Manna, Z. and A. Pnueli (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer.
- — (1995). *Temporal Verification of Reactive Systems: Safety*. Springer.
- McMillan, K. (July 2003). "Interpolation and SAT-based model checking". In: *Proc. 15th Int. Conf. on Computer Aided Verification (CAV 2003)*. Ed. by W. A. Hunt Jr. and F. Somenzi. Vol. 2725. LNCS. Boulder, CO, USA: Springer, pp. 1–13.
- Queille, J.-P. and J. Sifakis (Apr. 1982). "Specification and Verification of Concurrent Systems in CESAR". In: *Proc. 5th Int. Symp. Programming*. Ed. by M. Dezani-Ciancaglini and U. Montanari. Vol. 137. LNCS. Turin, Italy: Springer, pp. 337–351.
- Raymond, P. (July 1996). "Recognizing regular expressions by means of dataflow networks". In: *Proc. 23rd Int. Colloq. on Automata, Languages and Programming*. Ed. by F. Meyer auf der Heide and B. Monien. LNCS 1099. Paderborn, Germany: Springer, pp. 336–347.

References V

- Raymond, P., Y. Roux, and E. Jahier (2008). "Lutin: A Language for Specifying and Executing Reactive Scenarios". In: *EURASIP Journal of Embedded Systems*.
- Sheeran, M., S. Singh, and G. Stålmarck (Nov. 2000). "Checking Safety Properties Using Induction and a SAT-Solver". In: *Proc. 3rd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD 2000)*. Ed. by W. A. Hunt Jr. and S. D. Johnson. IEEE. Austin, TX, USA, pp. 127–144.
- Swamy, N., C. Hrițcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Zanella Béguelin (Jan. 2016). "Dependent Types and Multi-monadic Effects in F*". In: *Proc. 43rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2016)*. St. Petersburg, FL, USA: ACM Press, pp. 256–270.