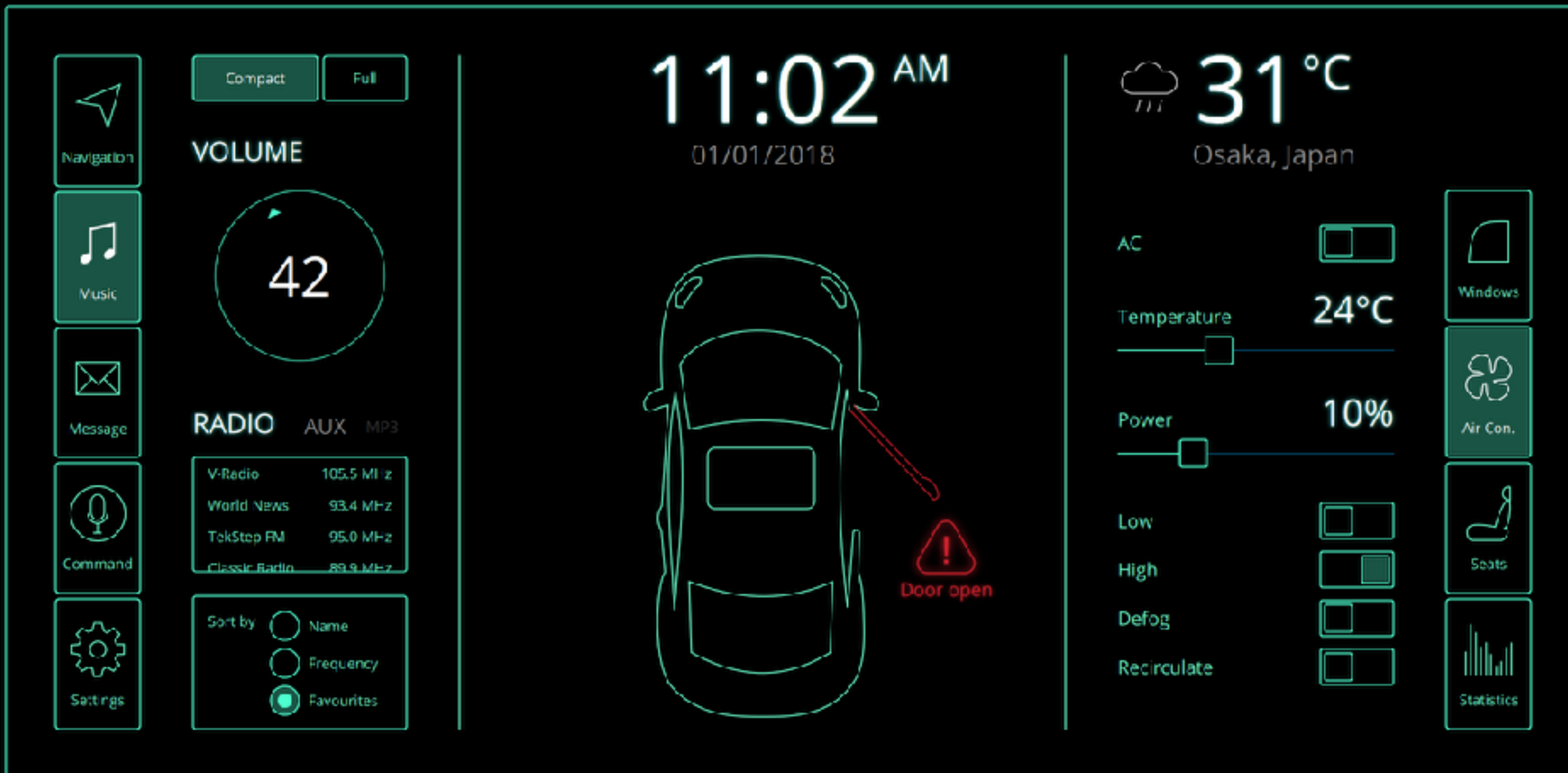


Formal Verification to Ensuring the Memory Safety of C++ Programs

Felipe R. Monteiro

Federal University of Amazonas
Institute of Computing
Postgraduate Programme in Informatics

Formal Verification to Ensuring the Memory Safety of C++ Programs

Master of Science in Informatics

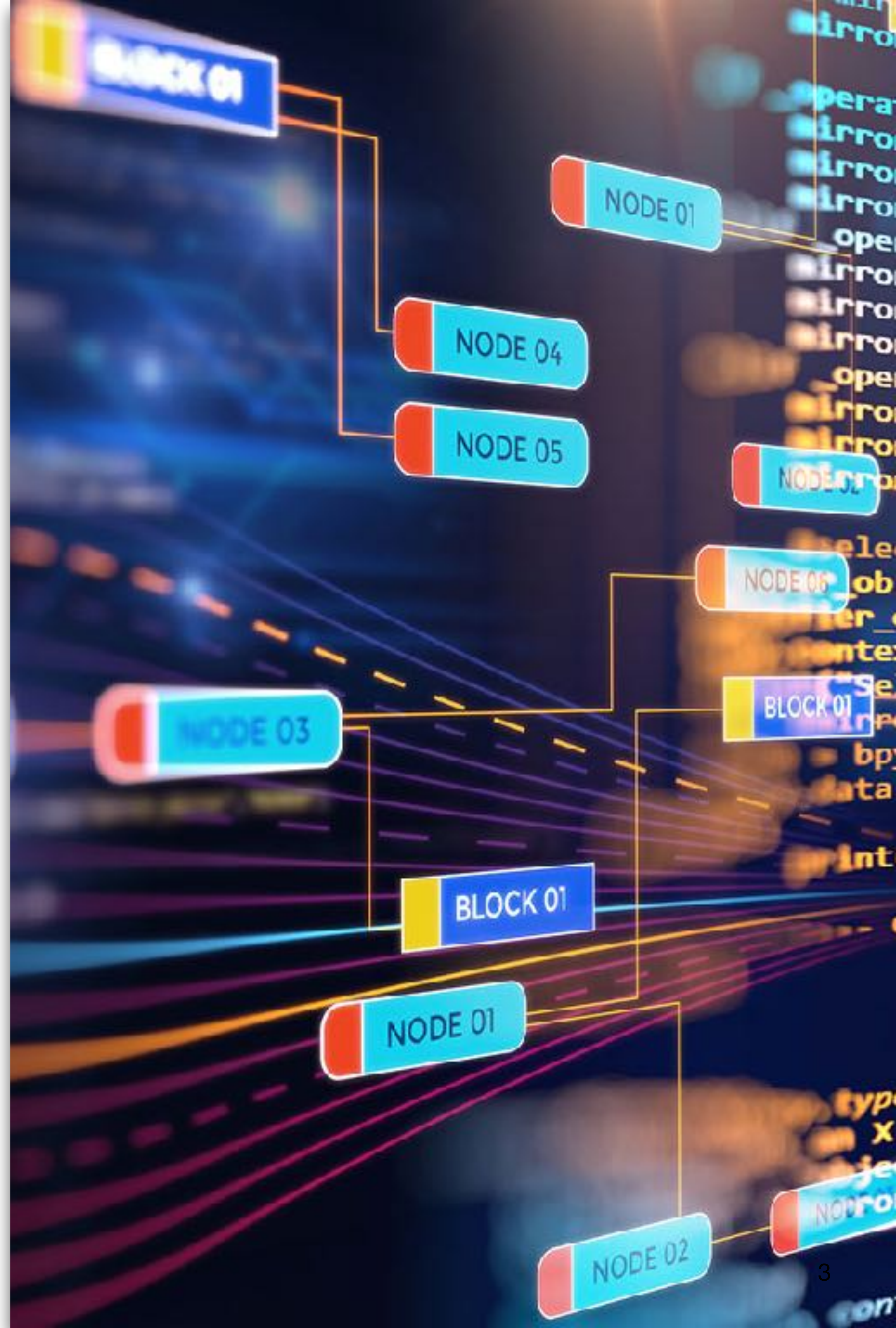
Felipe R. Monteiro
M.Sc. Candidate

Dr. Lucas C. Cordeiro
Supervisor

January 17, 2020
Manaus, Amazonas, Brazil

Problem & Motivation

Security is one of the most pressing issues of the 21st century

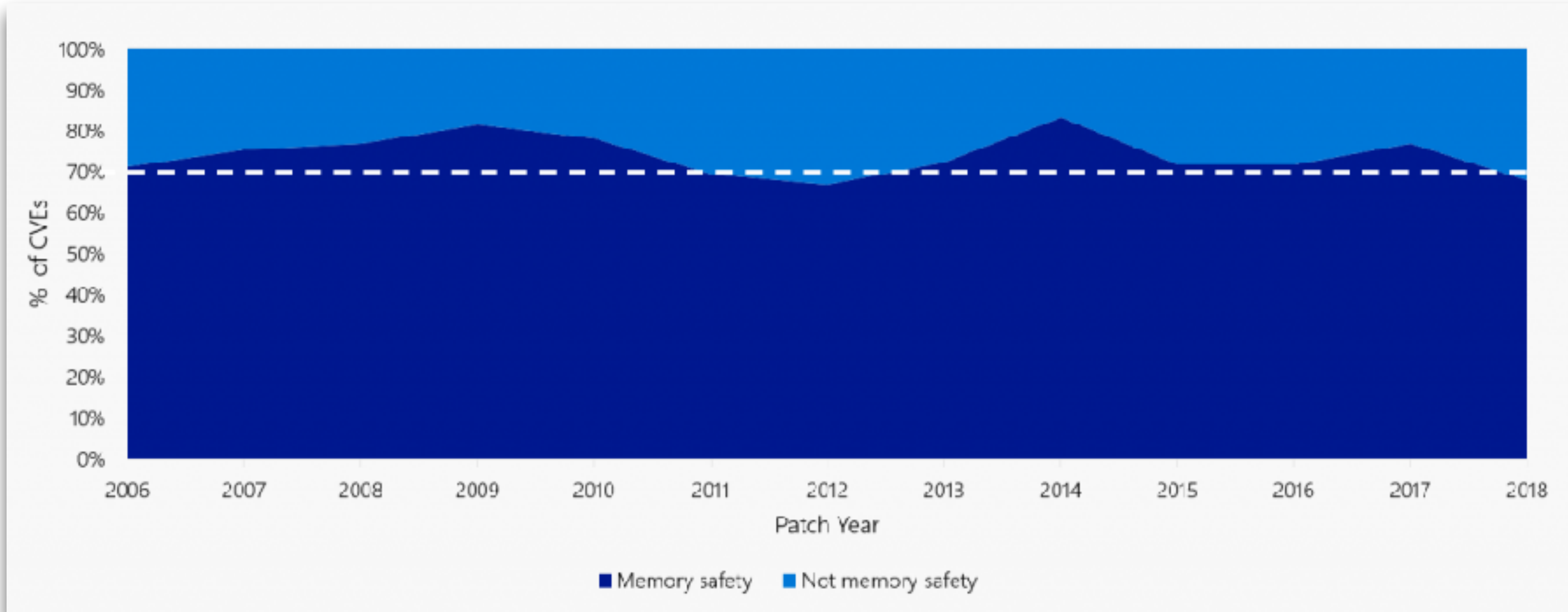


Consumer electronic products must be **as robust and bug-free as possible**, given that even medium product-return rates tend to be unacceptable

Consumer electronic products must be **as robust and bug-free as possible**, given that even medium product-return rates tend to be unacceptable

- “*Engineers reported the static analyser **Infer** was key to build a concurrent version of Facebook app to the Android platform.*”

Peter O’Hearn, FLoC, 2018.



- *"The majority of vulnerabilities are caused by developers inadvertently inserting* ***memory corruption bugs into their C and C++ code****. As Microsoft increases its code base and uses more Open Source Software in its code,* ***this problem isn't getting better, it's getting worse.****"*

Matt Miller, Microsoft Security Response Centre, 2019.

“*Formal automated reasoning is one of the investments that AWS is making in order to facilitate continued simultaneous growth in both functionality and security.*”

Byron Cook, FLoC, 2018.

"There has been a tremendous amount of valuable research in formal methods, but rarely have formal reasoning techniques been deployed as part of the development process of large industrial codebases."

facebook research

Peter O'Hearn, FLoC, 2018.

The research question is…

How to apply formal verification to ensuring memory safety of software written in the C++ programming language?

Main goal is to…

Apply model checking techniques to ensuring memory safety of C++ programs

Main goal is to…

Apply model checking techniques to ensuring memory safety of C++ programs

(i) Provide a **logical formalization of essential features** that the C++ programming language offers, such as templates, sequential and associative containers, inheritance, polymorphism, and exception handling.

Main goal is to…

Apply model checking techniques to ensuring memory safety of C++ programs

(i) Provide a **logical formalization of essential features** that the C++ programming language offers, such as templates, sequential and associative containers, inheritance, polymorphism, and exception handling.

(ii) Provide **a set of abstractions to the Standard C++ Libraries (SCL)** that reflects their semantics, in order **to enable the verification of functional properties** related to the use of these libraries.

Main goal is to…

Apply model checking techniques to ensuring memory safety of C++ programs

(i) Provide a **logical formalization of essential features** that the C++ programming language offers, such as templates, sequential and associative containers, inheritance, polymorphism, and exception handling.

(ii) Provide **a set of abstractions to the Standard C++ Libraries (SCL)** that reflects their semantics, in order **to enable the verification of functional properties** related to the use of these libraries.

(iii) **Extend an existing verifier to handle the verification of C++ programs** based on (i) and (ii) and evaluate its efficiency and effectiveness in comparison to similar state-of-the-art approaches.

Contributions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Contributions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Contributions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Contributions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Contributions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Contributions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Contributions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Background Theory

Satisfiability Module Theories,
Bounded Model Checking
& ESBMC Architecture

Satisfiability Modulo Theories

- Symbolic logic formula

`not x or (y and z)`

means

Either *x* is false **or** *y* and *z* are true (or both)

Satisfiability Modulo Theories

- Symbolic logic formula

```
not x or (y and z)
```

means

Either *x* is false **or** *y* and *z* are true (or both)

- Boolean Satisfiability (SAT)

```
not x or (y and z)
```

x = false, **y** = true, **z** = true

is **satisfiable**

Satisfiability Modulo Theories

- Symbolic logic formula

`not x or (y and z)`

means

Either *x* is false **or** *y* and *z* are true (or both)

- Boolean Satisfiability (SAT)

`not x or (y and z)`

x = false, **y** = true, **z** = true

is **satisfiable**

`not x and x`

is **unsatisfiable**

Satisfiability Modulo Theories

- As a generalisation of SAT, and the Boolean variables are replaced by other first-order theories:
  - Equality
  - Arithmetic
  - Arrays
  - Fixed-width bit-vectors
  - Inductive data types

$$x^2 - 4 = 0$$

x = 2

is **satisfiable**

Satisfiability Modulo Theories

- As a generalisation of SAT, and the Boolean variables are replaced by other first-order theories:
  - Equality
  - Arithmetic
  - Arrays
  - Fixed-width bit-vectors
  - Inductive data types

$$x^2 - 4 = 0$$

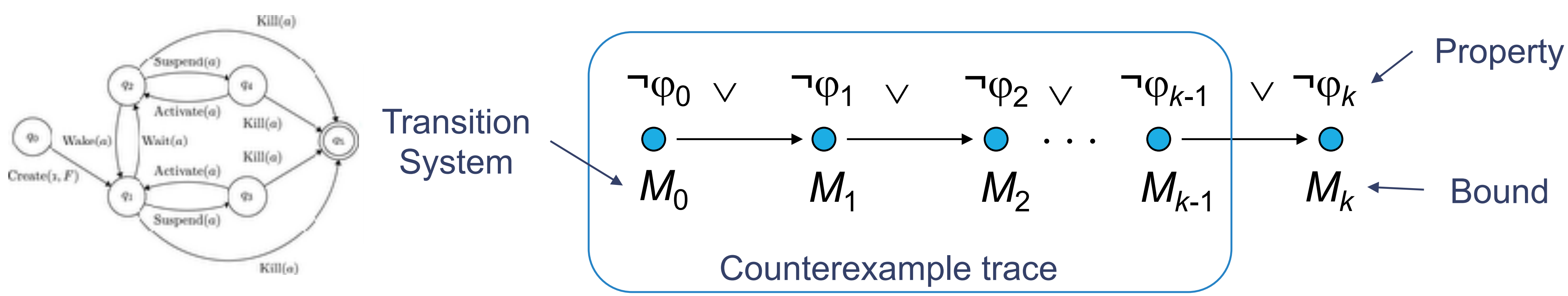
x = 2

is **satisfiable**

Where the key here is to take the problem and turn it into an SMT formula

Bounded Model Checking

- Basic Idea: given a transition system *M*, check negation of a given property ϕ up to given depth *k*



- Translated into a VC ψ such that: **ψ is satisfiable iff ϕ has counterexample of max. depth *k***
- BMC has been applied successfully to verify (embedded) software since early 2000's.

ESBMC Architecture

- **ESBMC** is an **open source**, permissively licensed, context-bounded model checker based on satisfiability modulo theories for the verification of single- and multi-threaded **C/C++ programs**.

- *It does not require the user annotates the programs with pre- or postconditions*, but allows the user to state additional properties using assert-statements, that are then checked as well.

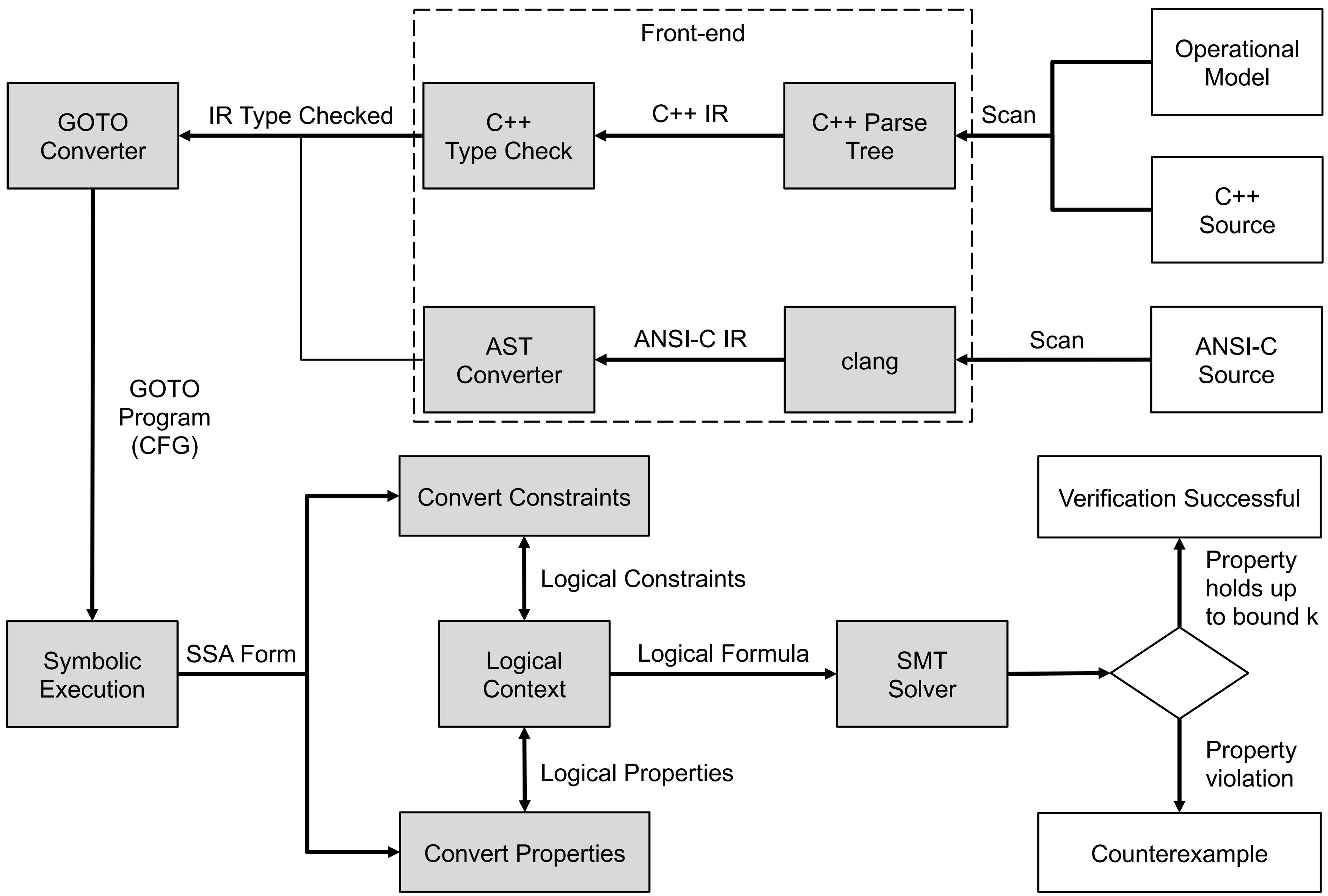
- It converts the verification conditions using different background theories and passes them directly to an SMT solver.

ESBMC is a joint project with
Federal University of Amazonas
University of Bristol
University of Manchester
University of Stellenbosch
University of Southampton

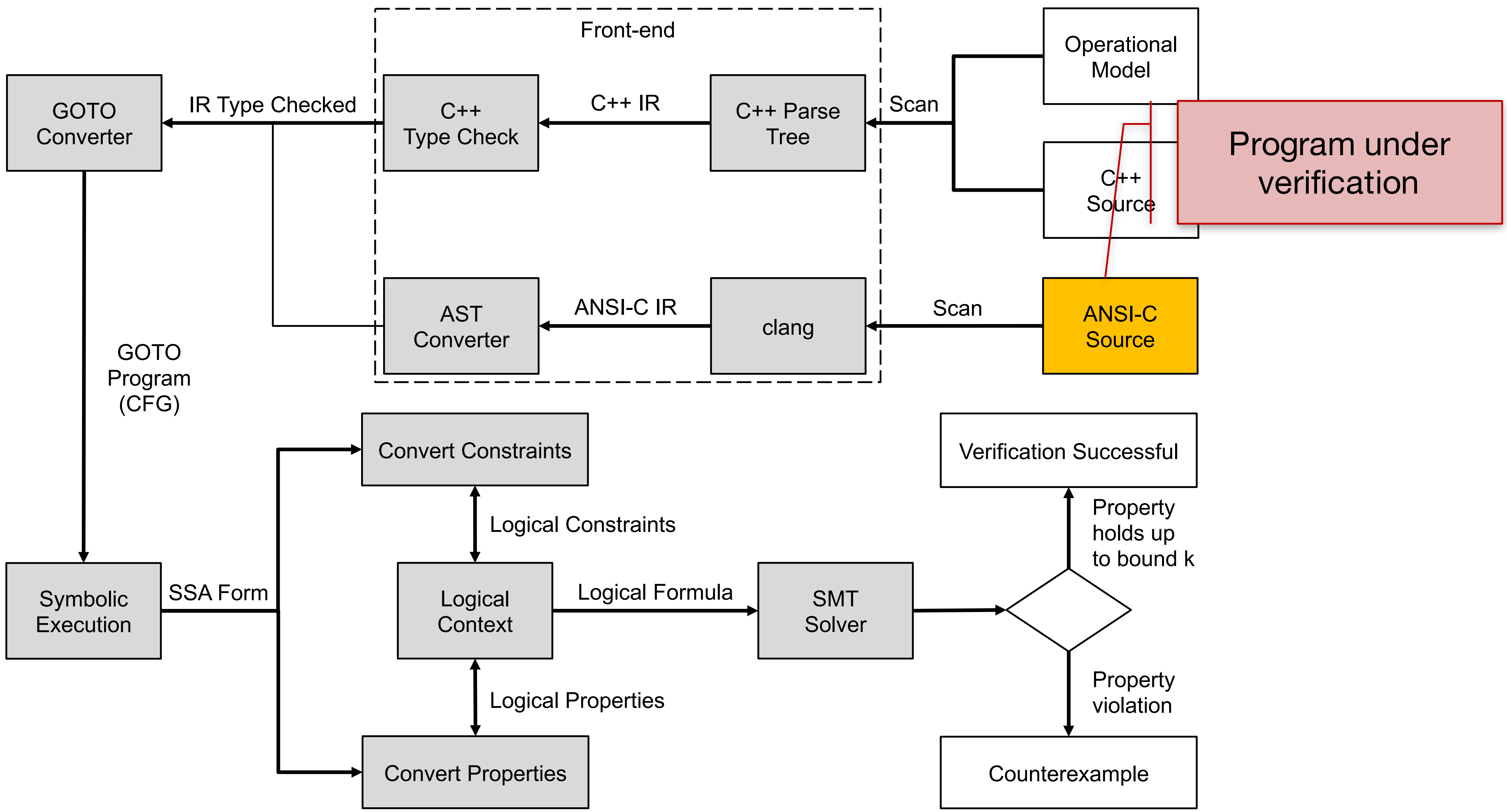
ESBMC

An Efficient SMT-based Bounded Model Checker.

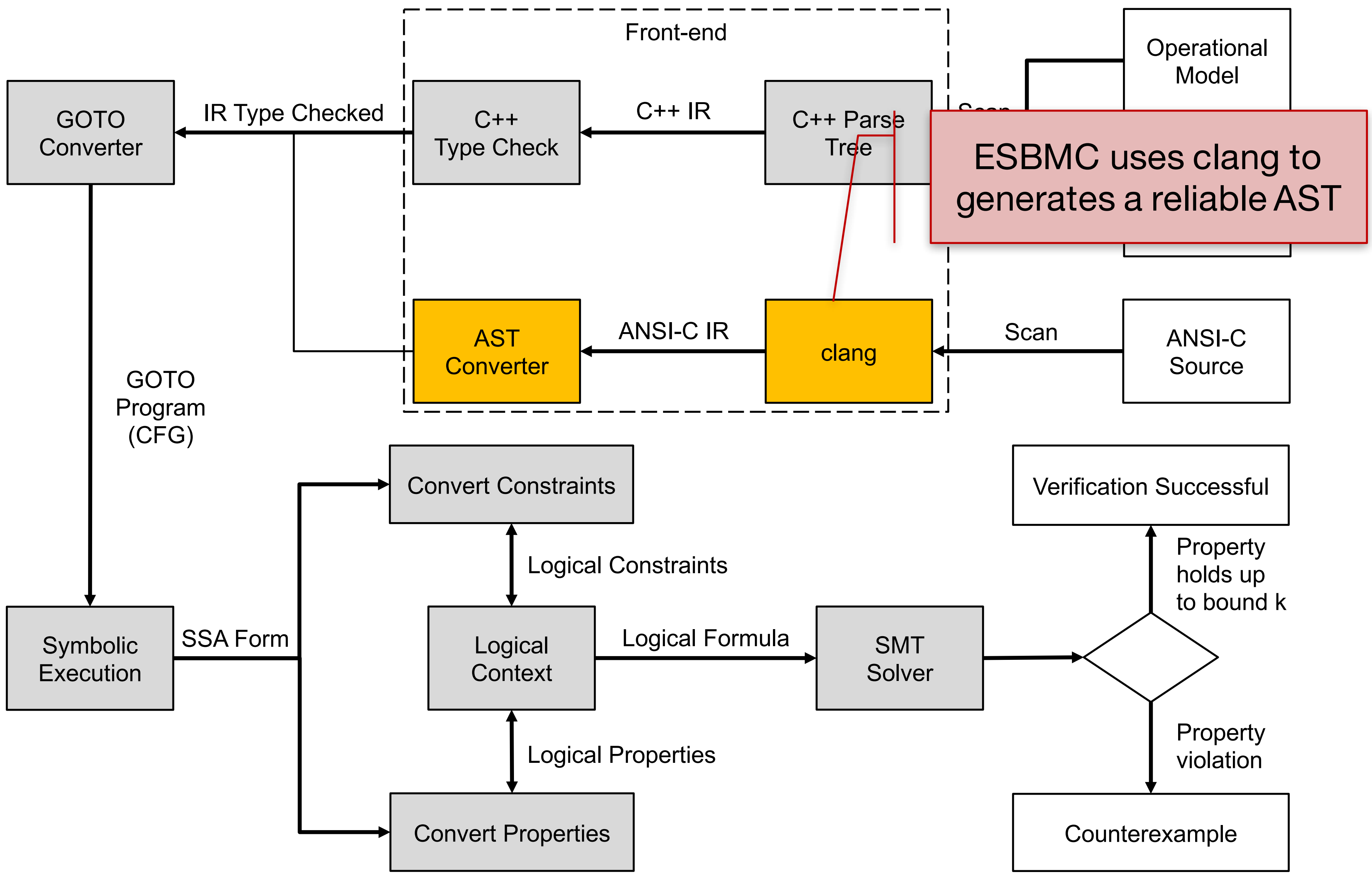
ESBMC Architecture



ESBMC Architecture

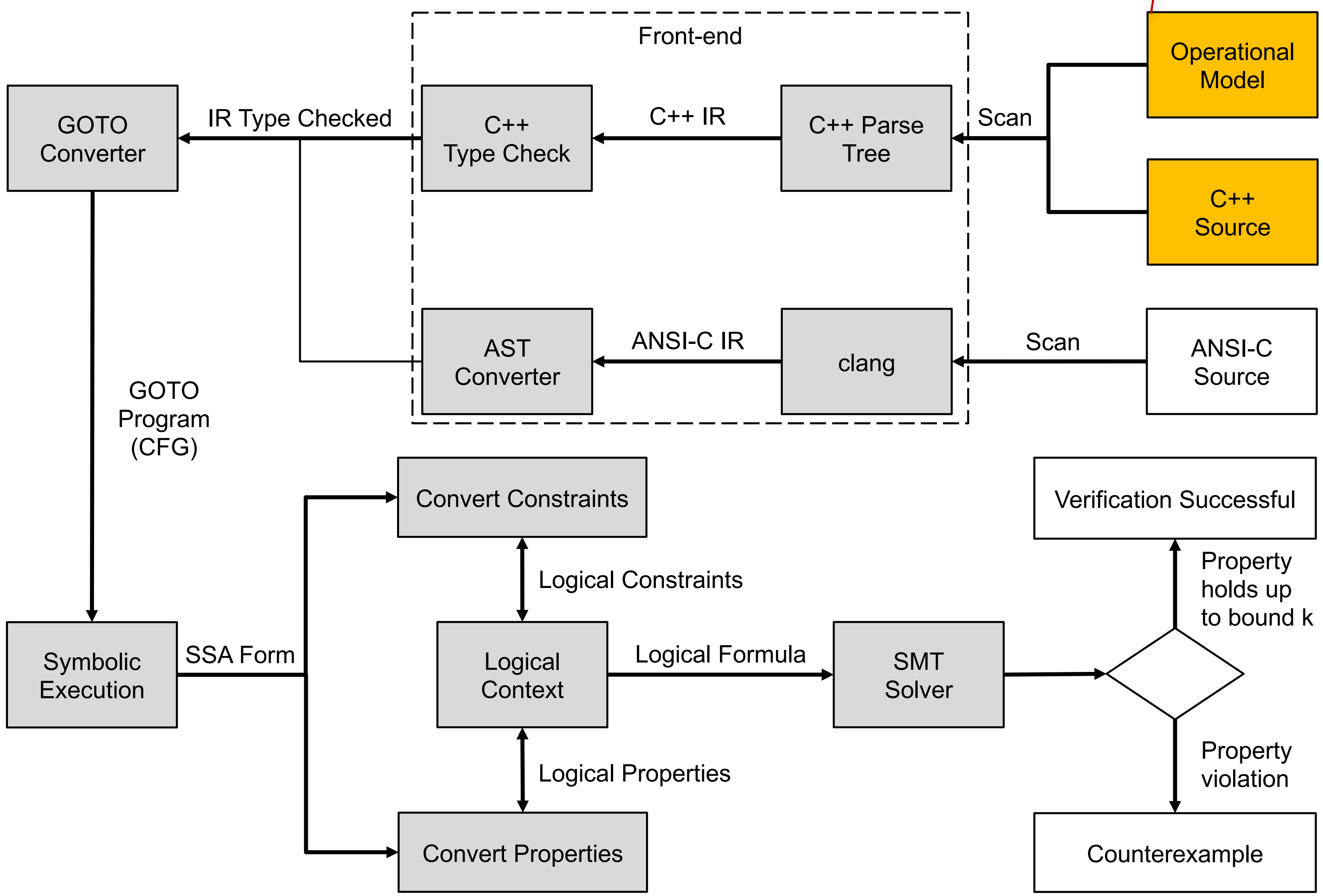


ESBMC Architecture

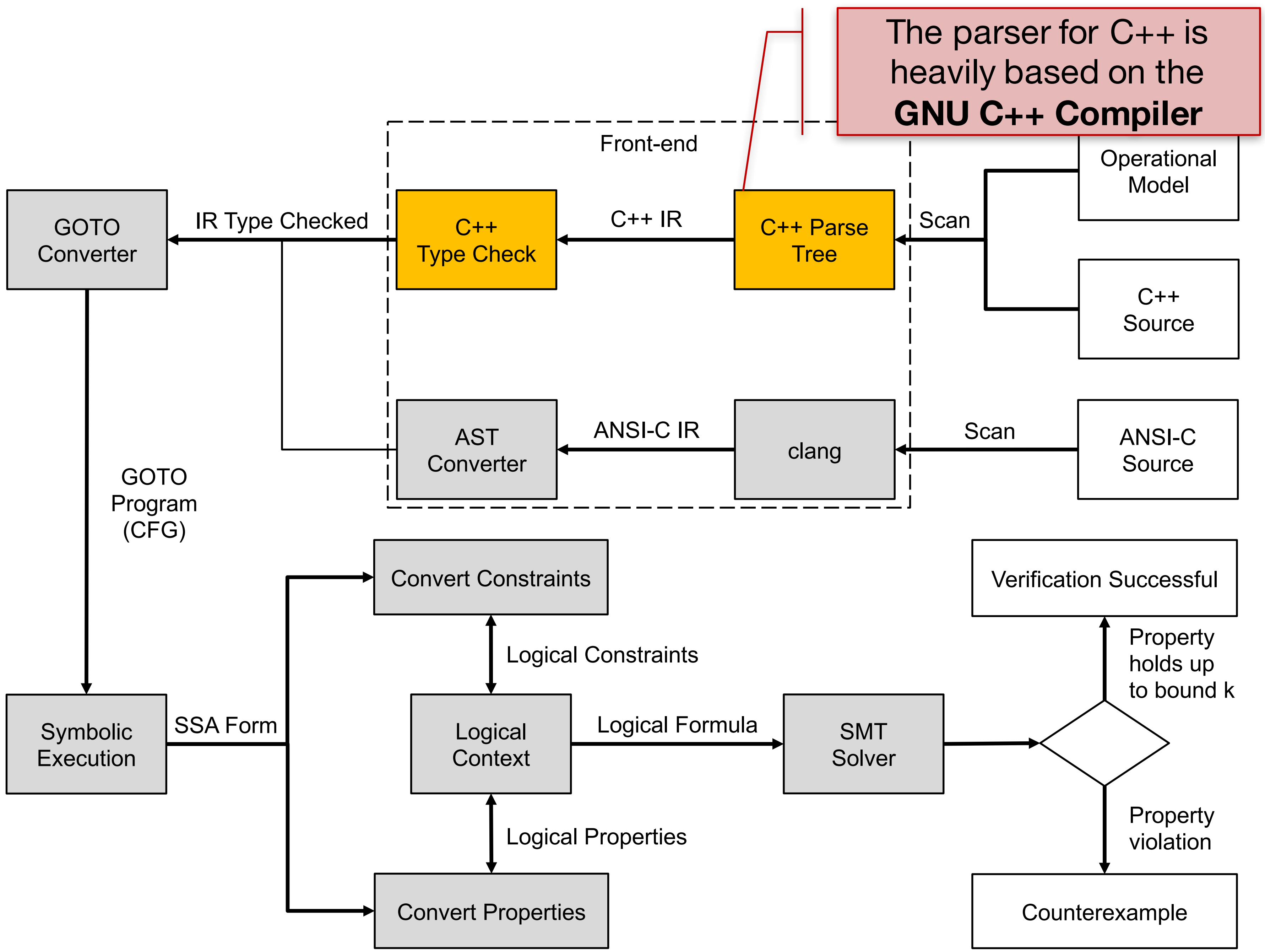


ESBMC Architecture

This is an additional extension for the ESBMC

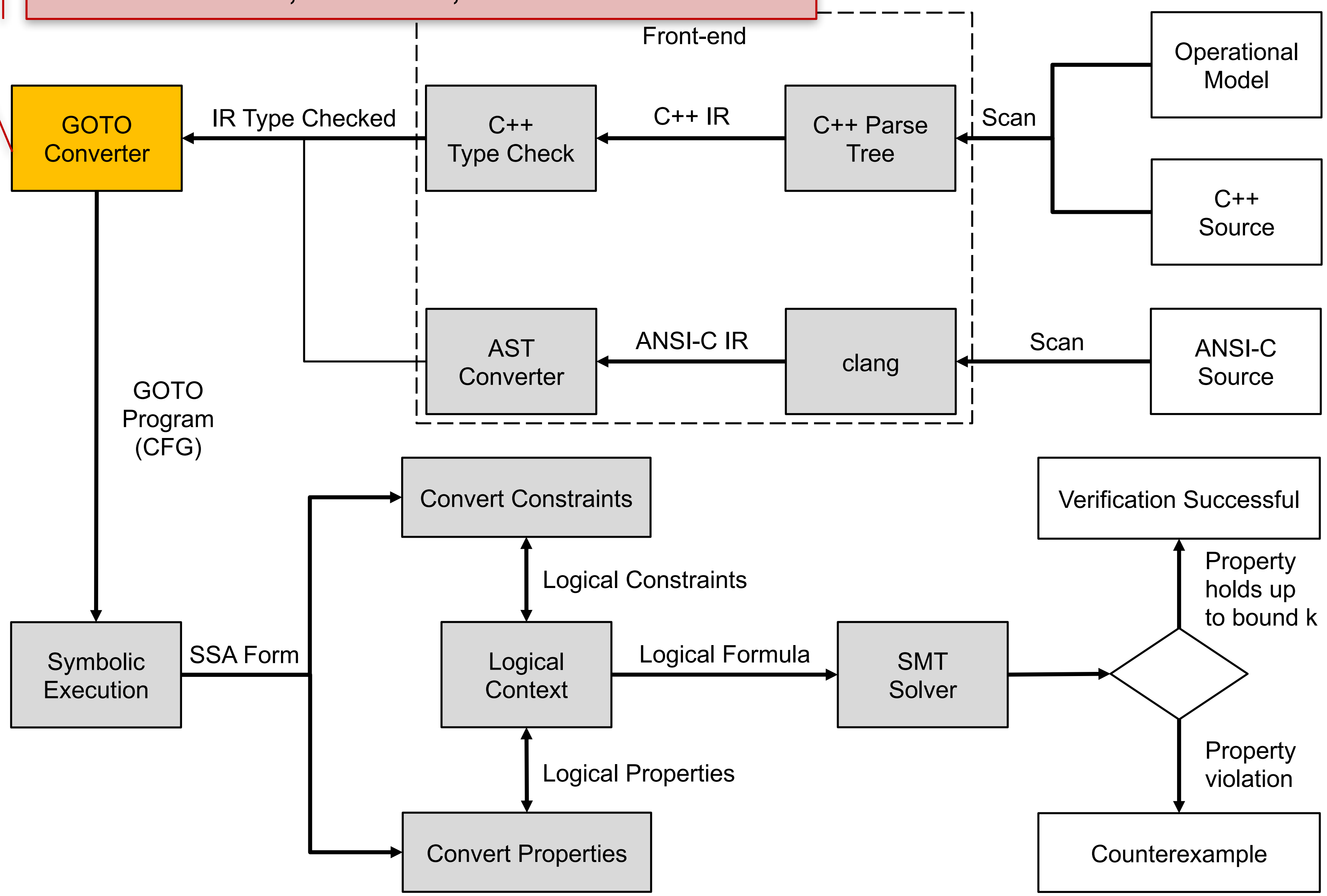


ESBMC Architecture

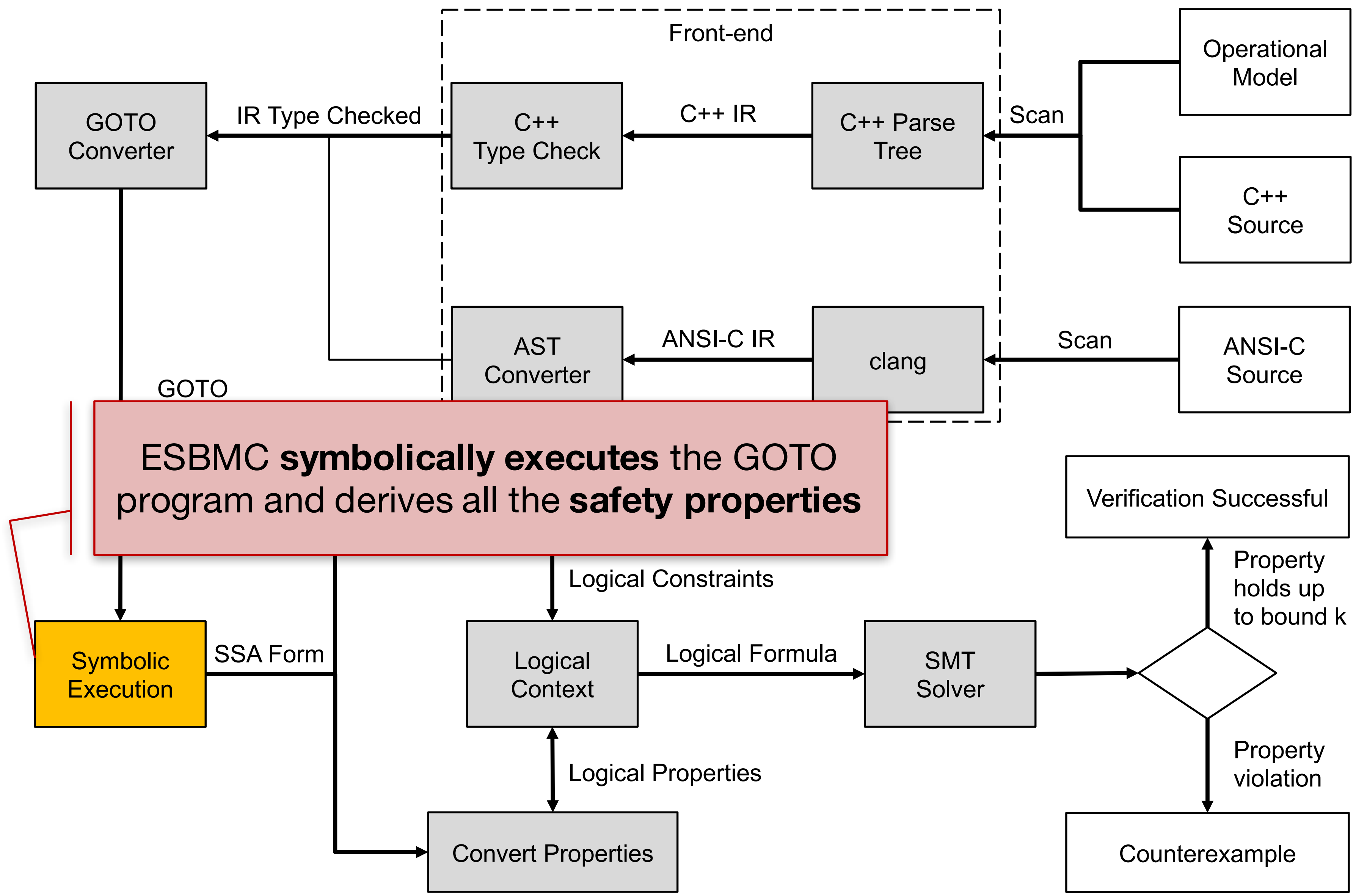


ESBMC Architecture

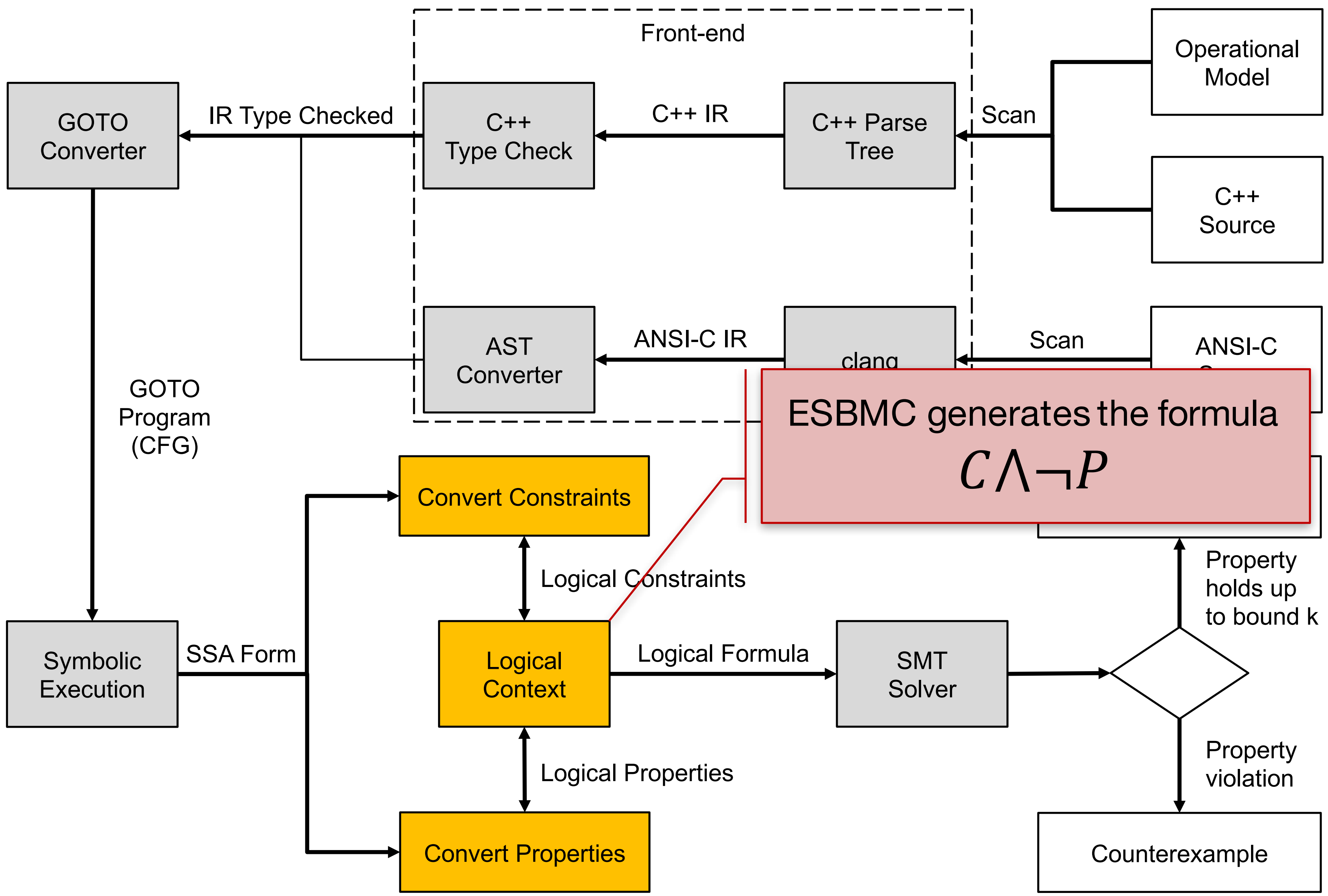
A **simplified representation** that consists only of assignments, conditional and unconditional branches, assumes, and assertions



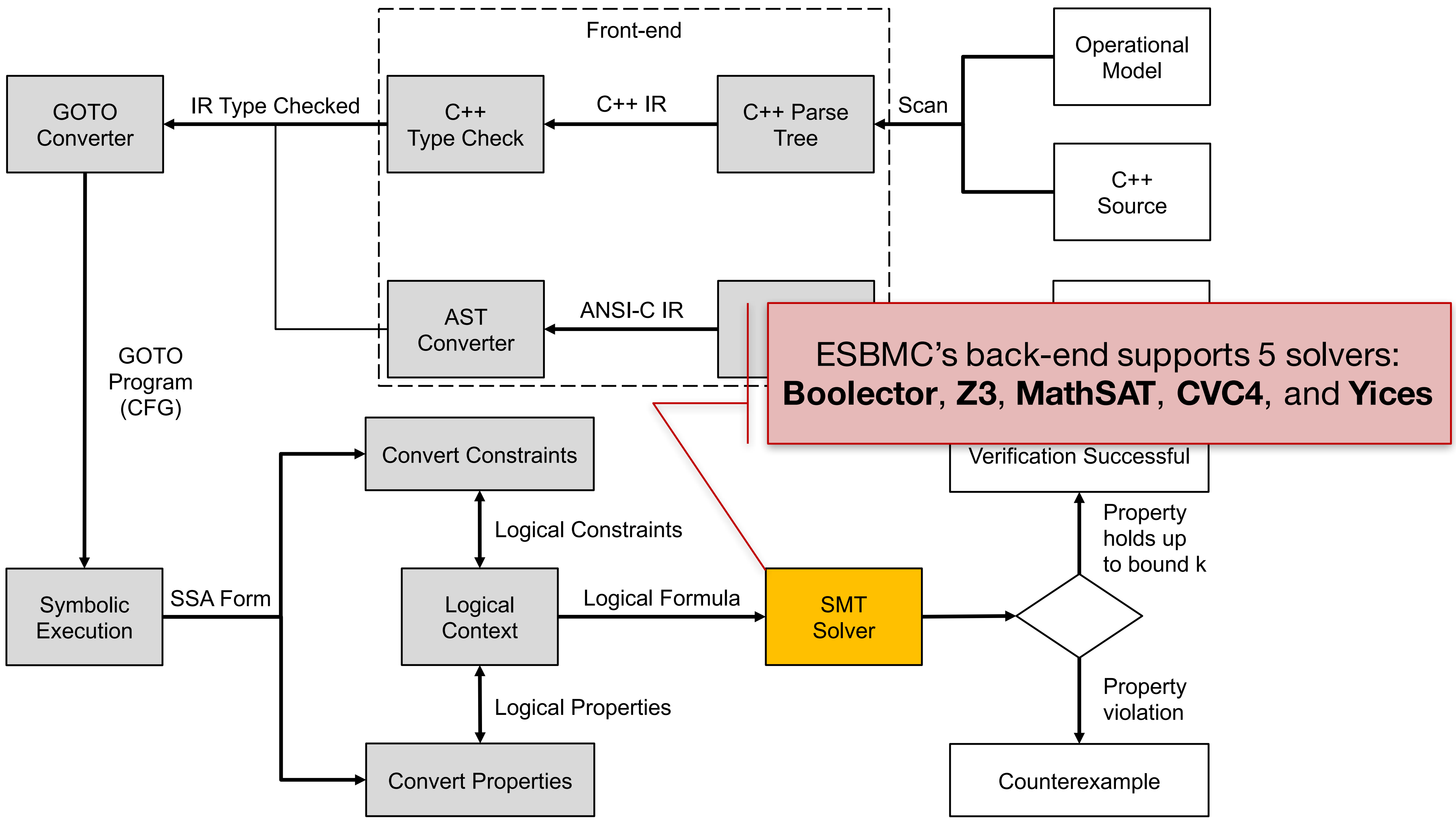
ESBMC Architecture



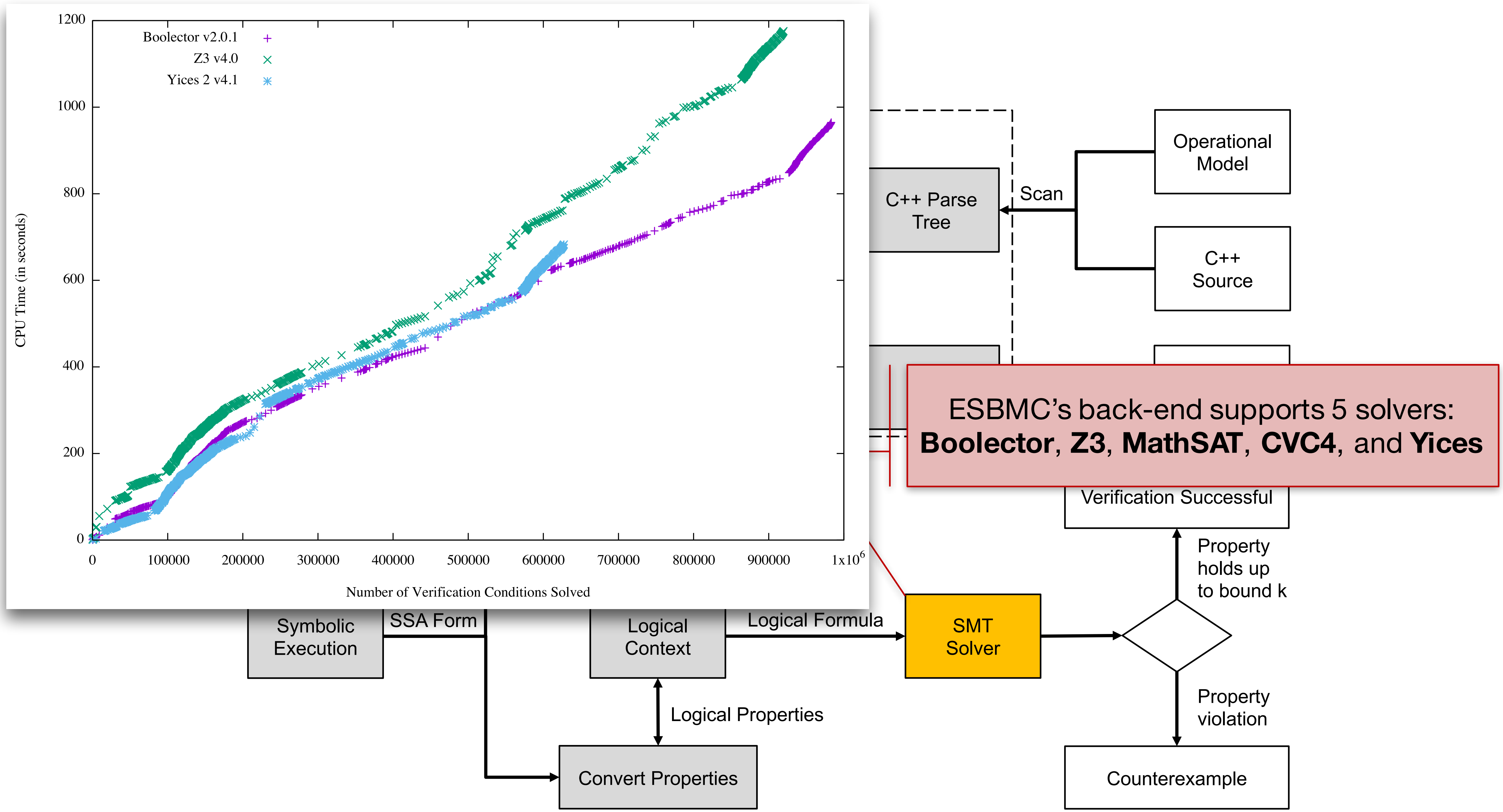
ESBMC Architecture



ESBMC Architecture

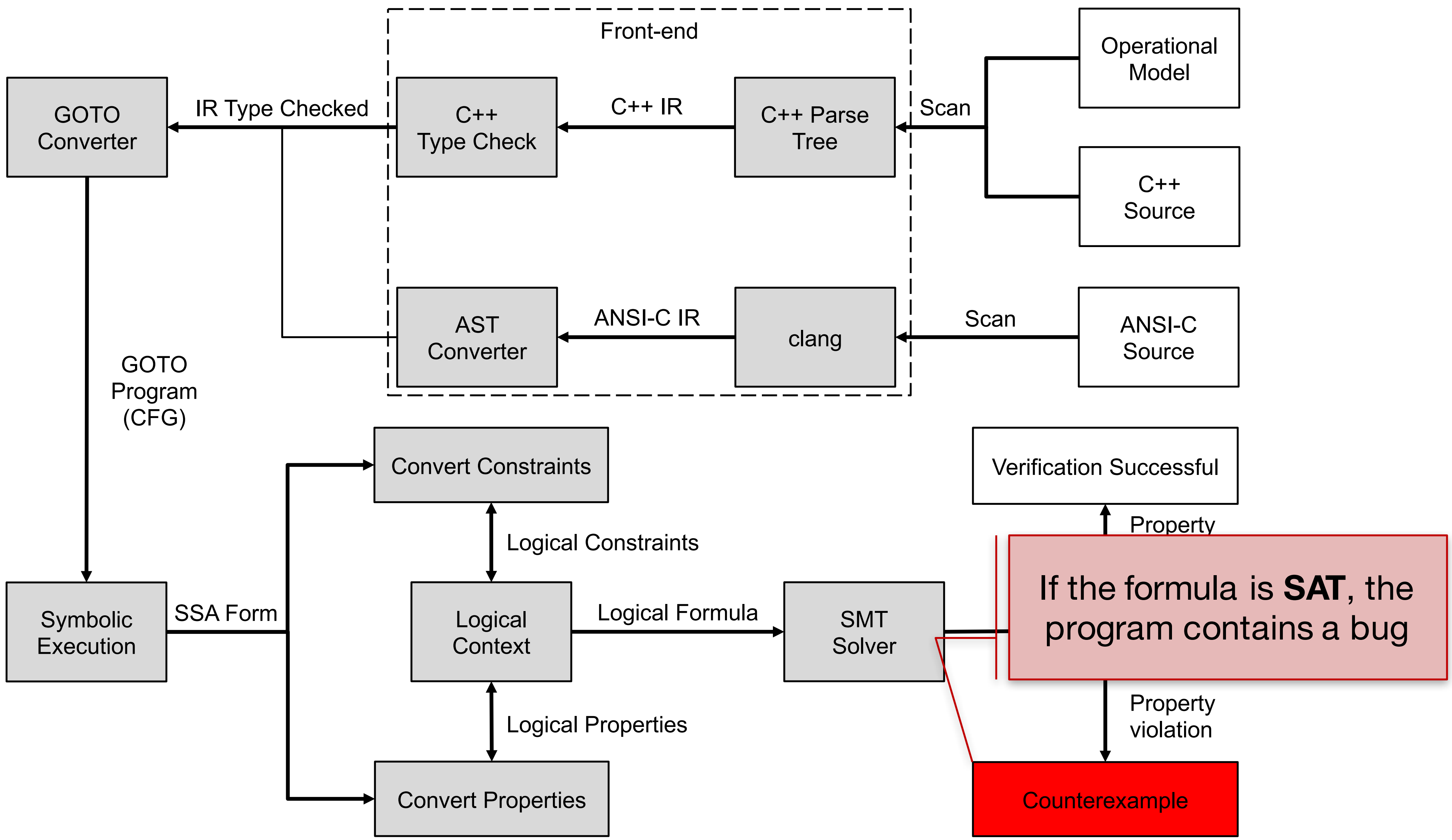


ESBMC Architecture

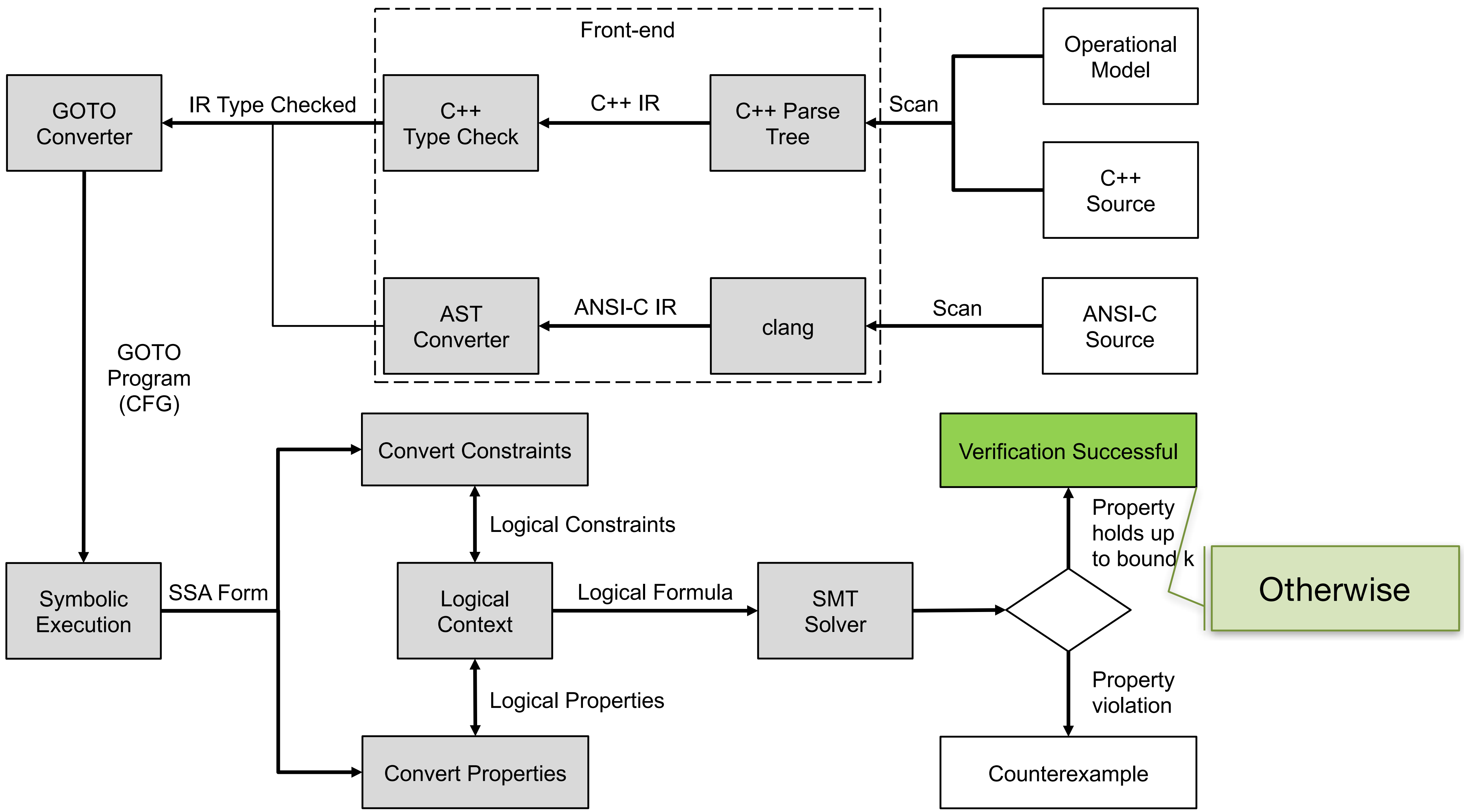


ESBMC's back-end supports 5 solvers: **Boolector**, **Z3**, **MathSAT**, **CVC4**, and **Yices**

ESBMC Architecture

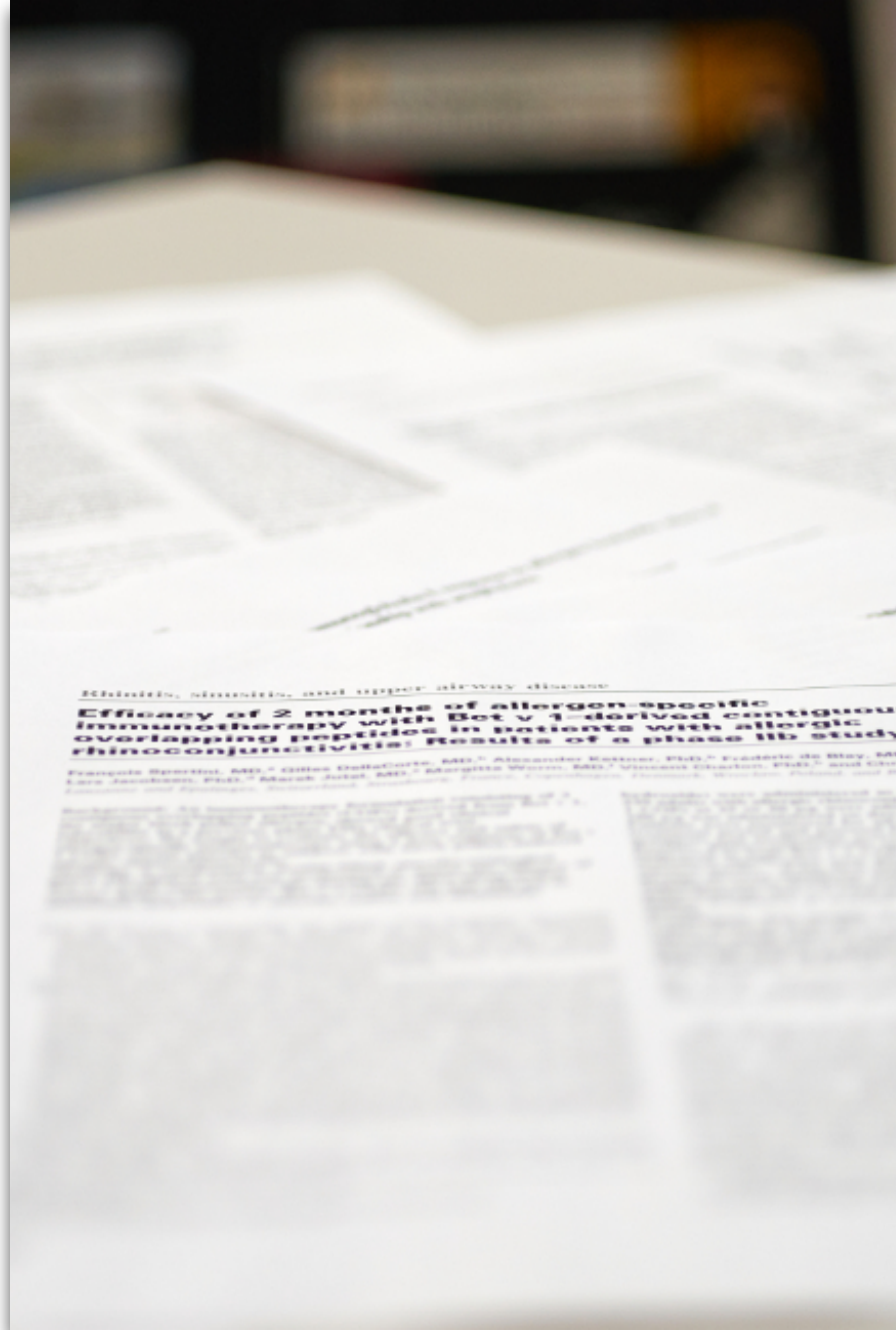


ESBMC Architecture



Related Work

What is out there?



Related Work

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC $v2.0$	**No**	**Yes**	**Yes**	**Yes**	**Yes**

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

Related Work

LLBMC

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC *v*2.0	**No**	**Yes**	**Yes**	**Yes**	**Yes**

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

Related Work

SATABS

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC $v2.0$	**No**	**Yes**	**Yes**	**Yes**	**Yes**

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

Related Work

CIL

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC *v*2.0	**No**	**Yes**	**Yes**	**Yes**	**Yes**

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

Related Work

CBMC

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC *v*2.0	**No**	**Yes**	**Yes**	**Yes**	**Yes**

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

Related Work

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC *v*2.0	**No**	**Yes**	**Yes**	**Yes**	**Yes**

DIVINE

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

Related Work

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC *v*2.0	**No**	**Yes**	**Yes**	**Yes**	**Yes**

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

Related Work

Related work	Conversion to other language	C++ Programming Language			
		Templates	Standard Template Libraries	Inheritance & Polymorphism	Exception Handling
Merz et al. [39]	LLVM	Yes	Yes	Yes	No
Blanc et al. [16]	No	Yes	Yes	No	No
Prabhu et al. [74]	ANSI-C	Yes	Not mentioned	Yes	Yes
Clarke et al. [24]	No	Yes	No	No	No
Baranová et al. [6]	LLVM	Yes	Yes	Yes	Yes
ESBMC *v*2.0	**No**	**Yes**	**Yes**	**Yes**	**Yes**

When it comes to the **verification of C++ programs**, most of the model checkers available focus on specific features

- Merz, Falke, and Sinz describe the **LLBMC** tool that uses BMC technique to verify C++ programs.

- Baranová et al. present **DIVINE**, an explicit-state model checker to verify single- and multi-threaded programs written in ANSI-C/C++

Approach and Uniqueness

SMT-based Bounded Model Checking of C++ Programs

SMT-based Bounded Model Checking C++ of Programs

Encoding essential features of C++ into SMT:

(i) *Primary template and explicit-template & partial-template specialization**

(ii) Standard Template Libraries

Sequential and Associative Containers

(iii) Inheritance & Polymorphism

(iv) *Exception Handling**

* *(R. Gadelha et al.)*

SMT-based Bounded Model Checking C++ of Programs

Encoding essential features of C++ into SMT:

(i) *Primary template and explicit-template & partial-template specialization**

(ii) Standard Template Libraries

Sequential and Associative Containers

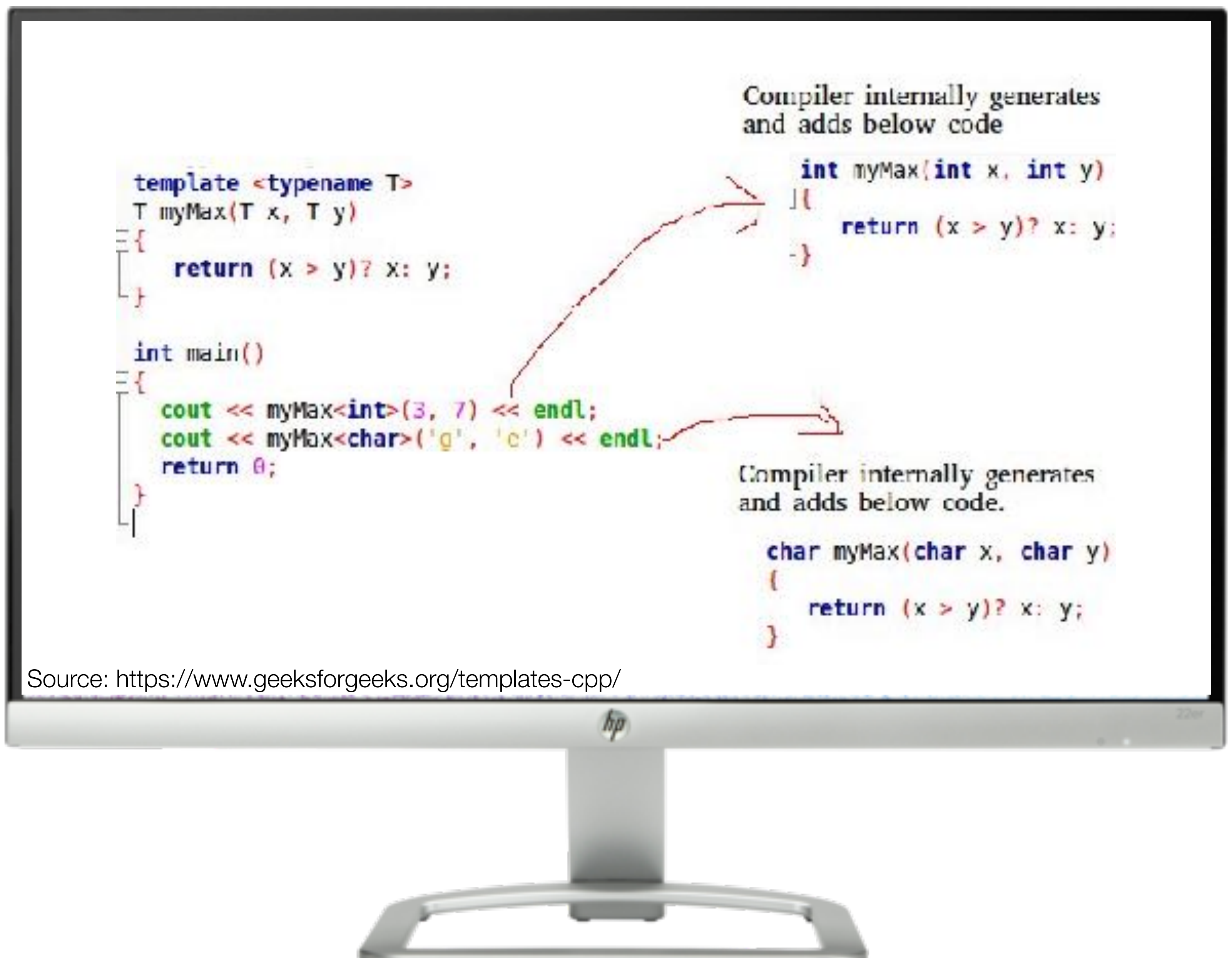
(iii) Inheritance & Polymorphism

(iv) *Exception Handling**

* *(R. Gadelha et al.)*

Templates

- Templates are used to define functions or classes of *generic data type*, which can be later instantiated with a specific data type



Source: https://www.geeksforgeeks.org/templates-cpp/

Templates

- Templates are used to define functions or classes of *generic data type*, which can be later instantiated with a specific data type

Reusability

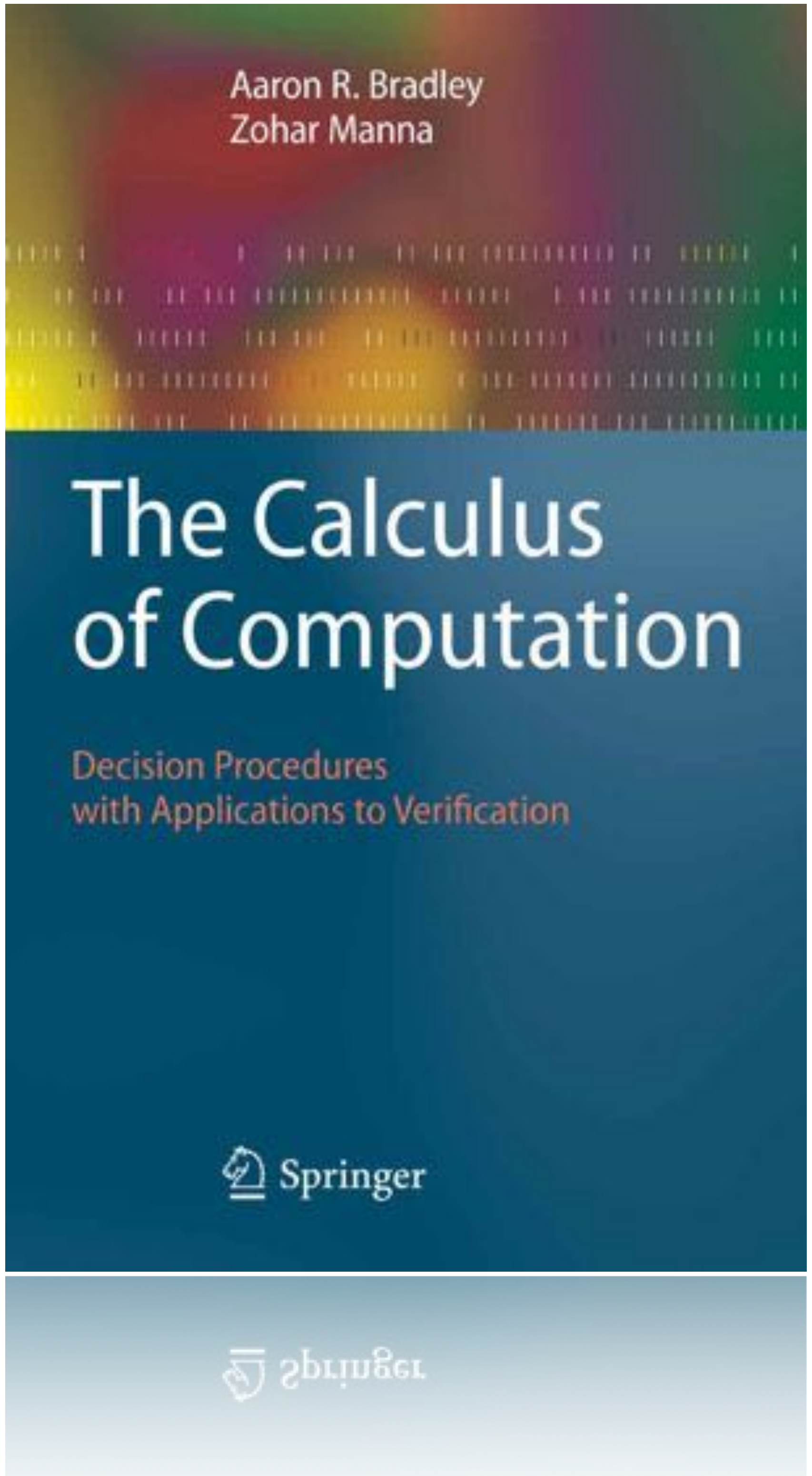
Templates

- Templates are used to define functions or classes of *generic data type*, which can be later instantiated with a specific data type

Reusability

- As described by Gadelha et al., in *ESBMC templates are only used until the type-checking phase*
  - At the end of the type-checking phase, all templates are discarded.

Templates


Aaron R. Bradley
Zohar Manna
The Calculus of Computation
Decision Procedures
with Applications to Verification
Springer

Templates

$$
\begin{aligned}
\Pi &:= \pi \\
T &:= \tau \\
S &:= s \mid s_e \mid s_p \\
\mathcal{A} &:= a \mid \mathbb{A} \\
\mathcal{N} &:= \texttt{name} \mid \mathcal{I}.\texttt{name} \mid G.\texttt{name} \\
\mathcal{K} &:= k \mid \mathcal{I}.k \mid G.k \mid \texttt{class} \mid \texttt{func}
\end{aligned}
$$

Templates

instantiations

$$
\begin{aligned}
\Pi &:= \pi \\
T &:= \tau \\
S &:= s \mid s_e \mid s_p \\
\mathcal{A} &:= a \mid \mathbb{A} \\
\mathcal{N} &:= \texttt{name} \mid \mathcal{I}.\texttt{name} \mid G.\texttt{name} \\
\mathcal{K} &:= k \mid \mathcal{I}.k \mid G.k \mid \texttt{class} \mid \texttt{func}
\end{aligned}
$$

Templates

instantiations

$$\Pi := \pi$$

templates

$$T := \tau$$

$$
\begin{aligned}
S &:= s \mid s_e \mid s_p \\
\mathcal{A} &:= a \mid \mathbb{A} \\
\mathcal{N} &:= \texttt{name} \mid \mathcal{I}.\texttt{name} \mid G.\texttt{name} \\
\mathcal{K} &:= k \mid \mathcal{I}.k \mid G.k \mid \texttt{class} \mid \texttt{func}
\end{aligned}
$$

Templates

$$\Pi := \pi$$

instantiations

$$T := \tau$$

templates

$$S := s \mid s_e \mid s_p$$

specializations

$$\mathcal{A} := a \mid \mathbb{A}$$

$$\mathcal{N} := \texttt{name} \mid \mathcal{I}.\texttt{name} \mid G.\texttt{name}$$

$$\mathcal{K} := k \mid \mathcal{I}.k \mid G.k \mid \texttt{class} \mid \texttt{func}$$

Templates

$$\Pi := \pi$$

instantiations

$$T := \tau$$

templates

$$S := s \mid s_e \mid s_p$$

specializations

$$\mathcal{A} := a \mid \mathbb{A}$$

arguments

$$\mathcal{N} := \texttt{name} \mid \mathcal{I}.\texttt{name} \mid G.\texttt{name}$$

$$\mathcal{K} := k \mid \mathcal{I}.k \mid G.k \mid \texttt{class} \mid \texttt{func}$$

Templates

$$\Pi := \pi$$

instantiations

$$T := \tau$$

templates

$$S := s \mid s_e \mid s_p$$

specializations

$$\mathcal{A} := a \mid \mathbb{A}$$

arguments

$$\mathcal{N} := \texttt{name} \mid \mathcal{I}.\texttt{name} \mid G.\texttt{name}$$

names

$$\mathcal{K} := k \mid \mathcal{I}.k \mid G.k \mid \texttt{class} \mid \texttt{func}$$

Templates

$$\Pi := \pi$$

instantiations

$$T := \tau$$

templates

$$S := s \mid s_e \mid s_p$$

specializations

$$\mathcal{A} := a \mid \mathbb{A}$$

arguments

$$\mathcal{N} := \mathtt{name} \mid \mathcal{I}.\mathtt{name} \mid G.\mathtt{name}$$

names

$$\mathcal{K} := k \mid \mathcal{I}.k \mid G.k \mid \mathtt{class} \mid \mathtt{func}$$

types

Templates

$$\mathcal{M}(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} \top, & \pi.\texttt{name} = \tau.\texttt{name} \land \pi.k = \tau.k \\ \bot, & \text{otherwise} \end{cases}$$

$$\lambda(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} s_{\texttt{M}}, & \forall \boldsymbol{s} \in \mathbb{S}_\tau \cdot (s_{\texttt{M}}, \mathbb{A}_\pi) \succeq (\boldsymbol{s}, \mathbb{A}_\pi) \\ \varnothing, & \text{otherwise} \end{cases}$$

Templates

match

$$\mathcal{M}(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} \top, & \pi.\mathtt{name} = \tau.\mathtt{name} \wedge \pi.k = \tau.k \\ \bot, & \text{otherwise} \end{cases}$$

$$\lambda(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} s_{\mathtt{M}}, & \forall s \in \mathbb{S}_\tau \cdot (s_{\mathtt{M}}, \mathbb{A}_\pi) \succeq (s, \mathbb{A}_\pi) \\ \varnothing, & \text{otherwise} \end{cases}$$

Templates

match

$$\mathcal{M}(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} \top, & \pi.\mathtt{name} = \tau.\mathtt{name} \wedge \pi.k = \tau.k \\ \bot, & \text{otherwise} \end{cases}$$

$$\lambda(\pi, \tau) \stackrel{\text{def}}{=} \begin{cases} s_{\mathtt{M}}, & \forall s \in \mathbb{S}_{\tau} \cdot (s_{\mathtt{M}}, \mathbb{A}_{\pi}) \succeq (s, \mathbb{A}_{\pi}) \\ \varnothing, & \text{otherwise} \end{cases}$$

Select the most specializated template

Templates

$$
\begin{aligned}
\mathcal{L}(\pi, \tau_1, ..., \tau_q) := ite(\mathcal{M}(\pi, \tau_1), \tau_\pi = \tau_1,\\
& ite(\mathcal{M}(\pi, \tau_2), \tau_\pi = \tau_2,\\
& \dots\\
& ite(\mathcal{M}(\pi, \tau_q), \tau_\pi = \tau_q, \tau_\pi = \varnothing) \dots)\\
& \wedge \tau_\pi \neq \varnothing\\
& \wedge s = \lambda(\pi, \mathbb{S}_{\tau_\pi})\\
& \wedge ite(s = \varnothing, \tau_\pi, s)
\end{aligned}
$$

Templates

```
#include<cassert>
using namespace std;

// template creation
template <typename T>
bool qCompare(const T a, const T b) {
  return (a > b) ? true : false;
}

template <typename T>
bool qCompare(T a, T b) {
  return (a > b) ? true : false;
}

// template specialization
template<>
bool qCompare(float a, float b) {
  return (b > a) ? true : false;
}

int main() {
  // template instantiation
  assert((qCompare(1.5f, 2.5f)));
  assert((qCompare<int>(1, 2) == false));
  return 0;
}
```

Templates

```
#include<cassert>
using namespace std;

// template creation
template <typename T>
bool qCompare(const T a, const T b) {
  return (a > b) ? true : false;
}

template <typename T>
bool qCompare(T a, T b) {
  return (a > b) ? true : false;
}

// template specialization
template<>
bool qCompare(float a, float b) {
  return (b > a) ? true : false;
}

int main() {
  // template instantiation
  assert((qCompare(1.5f, 2.5f)));
  assert((qCompare<int>(1, 2) == false));
  return 0;
}
```

Template creation

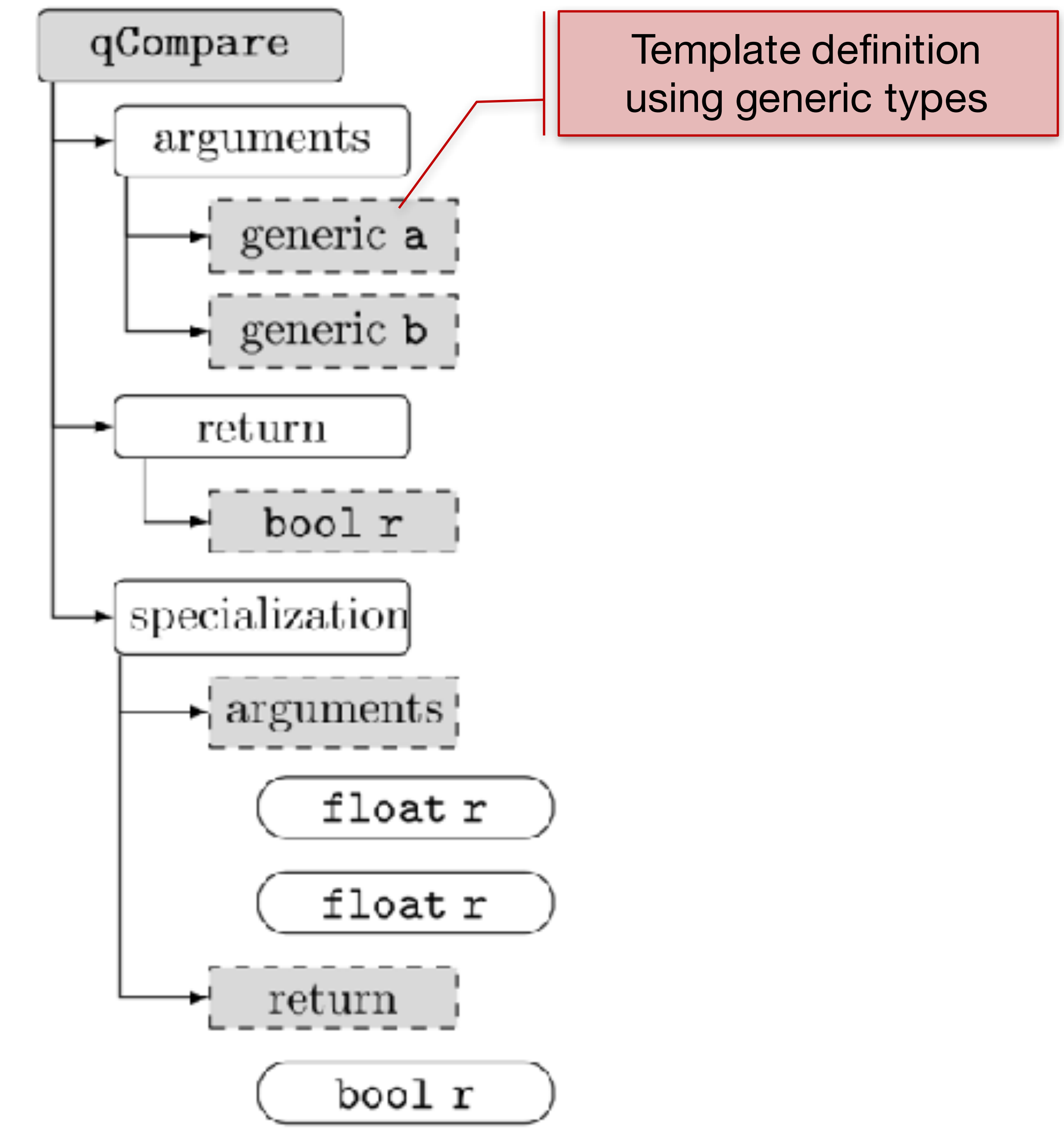
Templates

```
#include<cassert>
using namespace std;

// template creation
template <typename T>
bool qCompare(const T a, const T b) {
   return (a > b) ? true : false;
}

template <typename T>
bool qCompare(T a, T b) {
   return (a > b) ? true : false;
}

// template specialization
template<>
bool qCompare(float a, float b) {
   return (b > a) ? true : false;
}

int main() {
   // template instantiation
   assert((qCompare(1.5f, 2.5f)));
   assert((qCompare<int>(1, 2) == false));
   return 0;
}
```

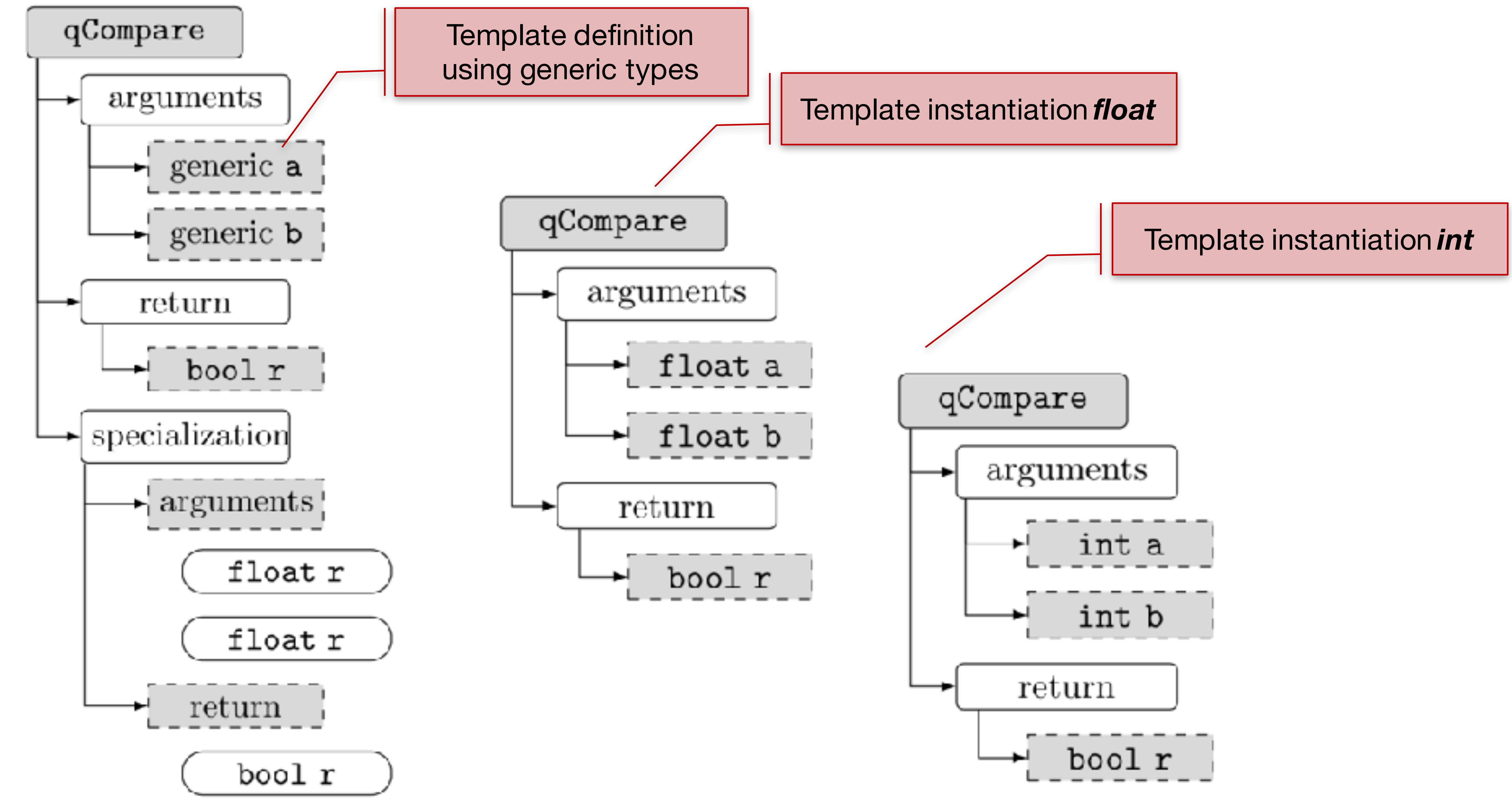
Template instantiation ***float***

Template instantiation ***int***

Templates


qCompare
arguments
generic a
generic b
return
bool r
specialization
arguments
float r
float r
return
bool r
Template definition
using generic types

Templates


qCompare
arguments
generic a
generic b
return
bool r
specialization
arguments
float r
float r
return
bool r
Template definition using generic types
Template instantiation *float*
qCompare
arguments
float a
float b
return
bool r
Template instantiation *int*
qCompare
arguments
int a
int b
return
bool r

Templates

SSA Form

```
a1 = 1.5f
b1 = 2.5f
return_qcompare1 = (b1 > a1)? TRUE : FALSE
a2 = 1
b2 = 2
return_qcompare2 = (a2 > b2)? TRUE : FALSE
```

Templates

$$\mathcal{C} := \left[\begin{array}{l} a_1 = 1.5f \wedge b_1 = 2.5f \\ \wedge\ return\_qcompare_1 = ite\,(b_1 > a_1, 1, 0) \\ \wedge\ a_2 = 1 \wedge b_2 = 2 \\ \wedge\ return\_qcompare_2 = ite\,(a_2 > b_2, 1, 0) \end{array}\right]$$

$$\mathcal{P} := \left[\begin{array}{l} return\_qcompare_1 = \top \\ \wedge\ return\_qcompare_2 = \bot \end{array}\right]$$

SMT-based Bounded Model Checking C++ of Programs

Encoding essential features of C++ into SMT:

(i) *Primary template and explicit-template & partial-template specialization**

(ii) Standard Template Libraries

Sequential and Associative Containers

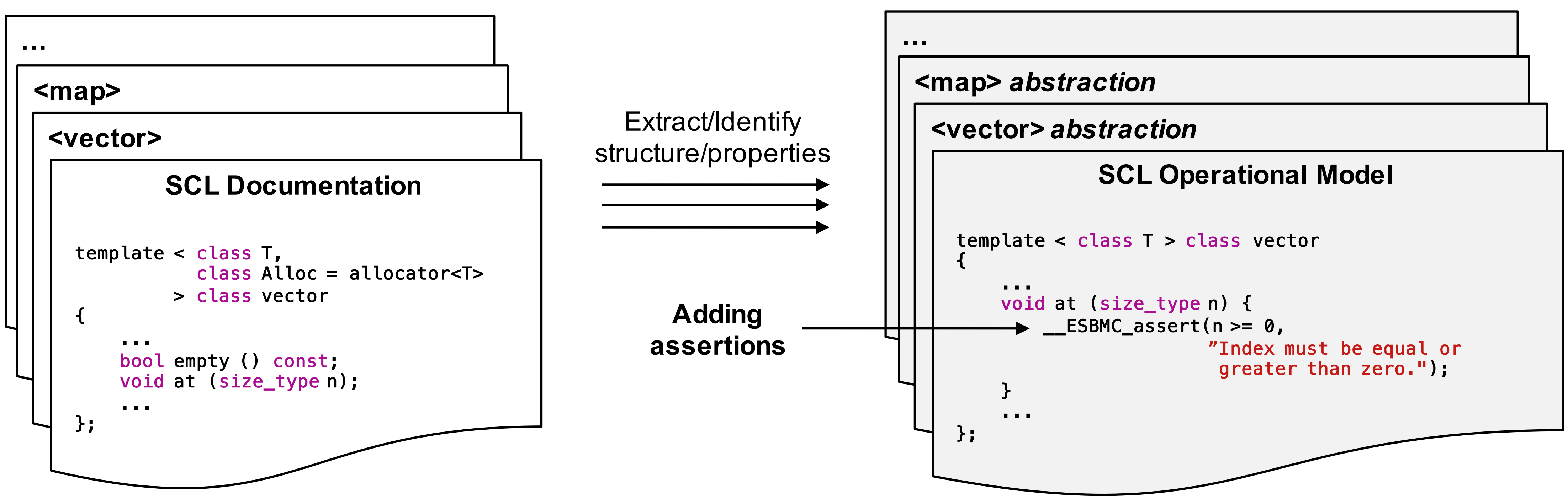
(iii) Inheritance & Polymorphism

(iv) *Exception Handling**

* *(R. Gadelha et al.)*

Building Operational Models

- We base the development process of operational models in the **documentation** of the Standard C++ Library
  - the operational model is an abstract representation, which is used to identify elements and verify specific properties related to C++ libraries

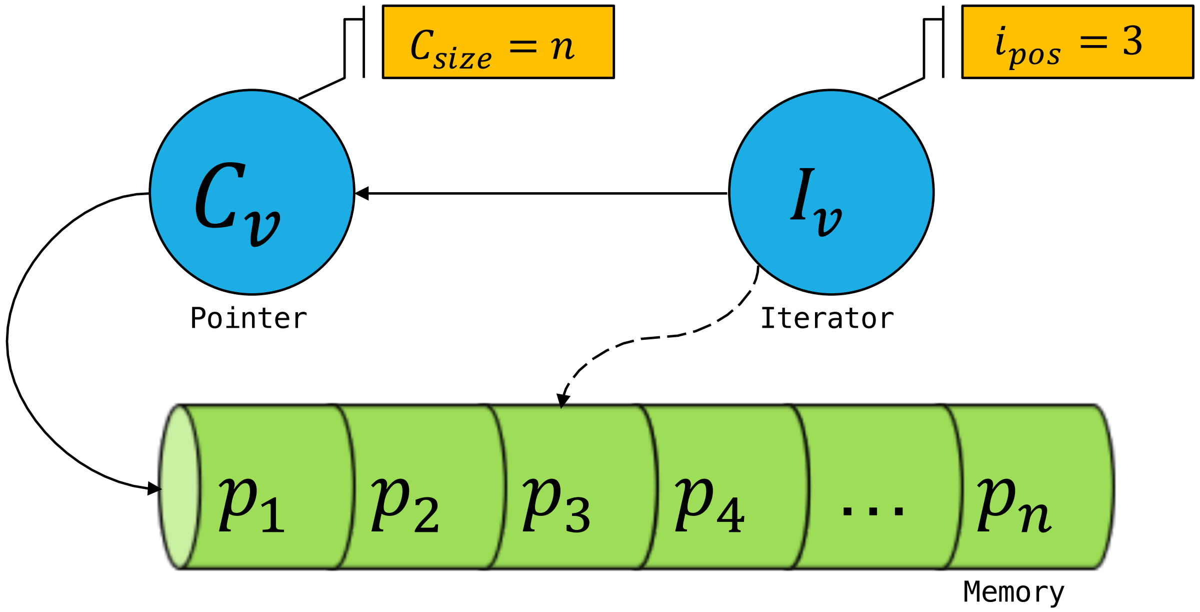


Operational Models for Containers

"*The Containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures*"

cppreference.com, 2018.

- **Sequential containers** are built into a structure to store elements of a certain type $\mathbf{V}$, in a certain sequential order.
- Note that all methods, from those libraries, can be expressed as simplified variations of 3 main operations:
  - insertion $\mathbf{C.insert\ (I,\ V,\ N)}$
  - deletion $\mathbf{C.erase\ (I)}$
  - search $\mathbf{C.search\ (V)}$

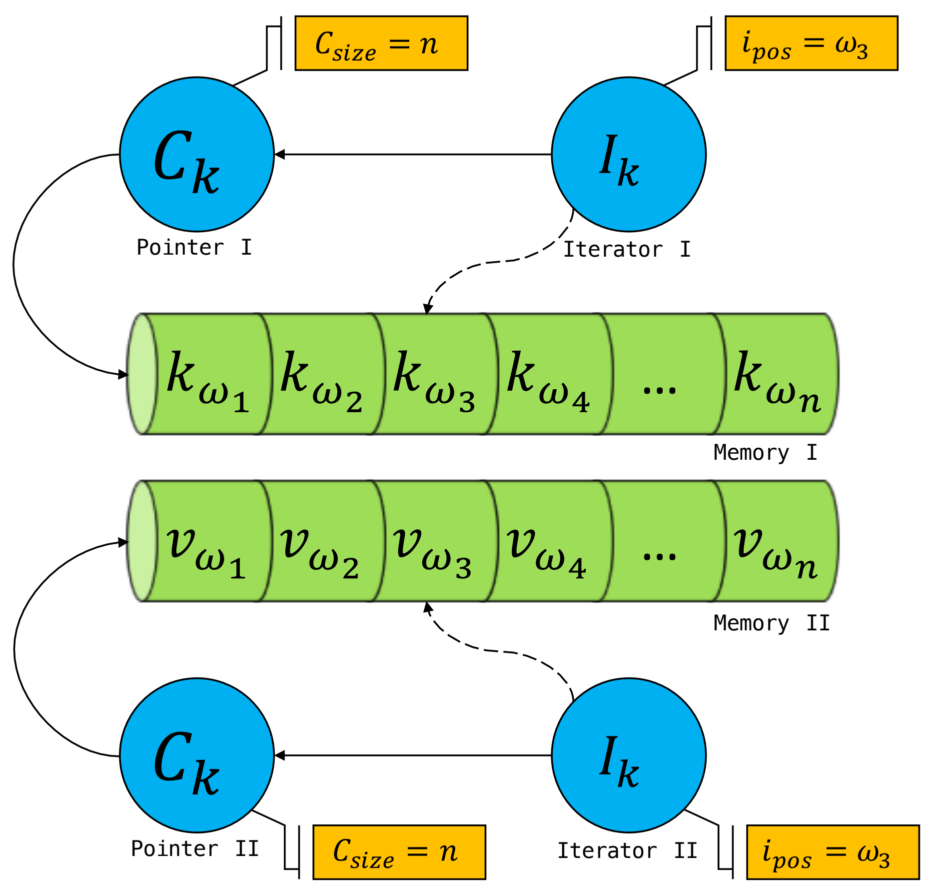


Operational Models for Containers

"*The Containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures*"

cppreference.com, 2018.

- **Associative containers** connects each key, of a certain type $\mathbf{K}$, to a value, of a certain type $\mathbf{V}$, where associated keys are stored in order.
- Note that all methods, from those libraries, can be expressed as simplified variations of three main operations:
  - insertion $\mathrm{C.insert\ (I,\ V,\ N)}$
  - deletion $\mathrm{C.erase\ (I)}$
  - search $\mathrm{C.search\ (K)}$



SMT-based Bounded Model Checking C++ of Programs

Encoding essential features of C++ into SMT:

(i) *Primary template and explicit-template & partial-template specialization**

(ii) Standard Template Libraries

Sequential and Associative Containers

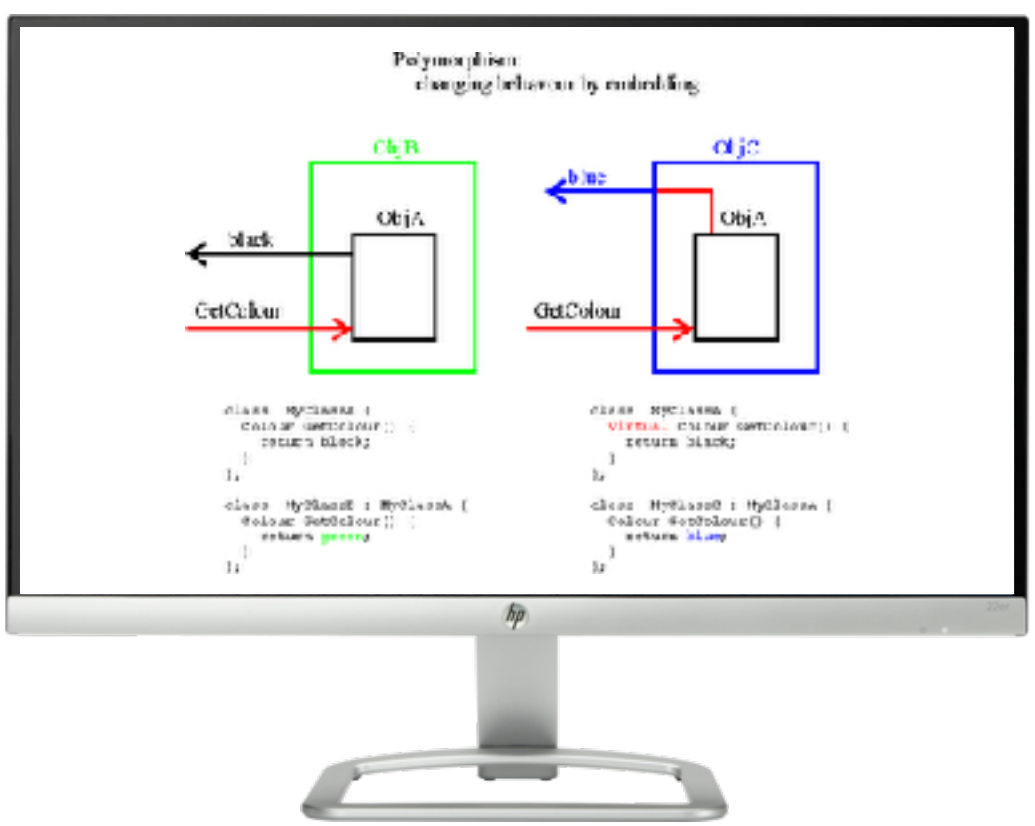
(iii) Inheritance & Polymorphism

(iv) *Exception Handling**

* *(R. Gadelha et al.)*

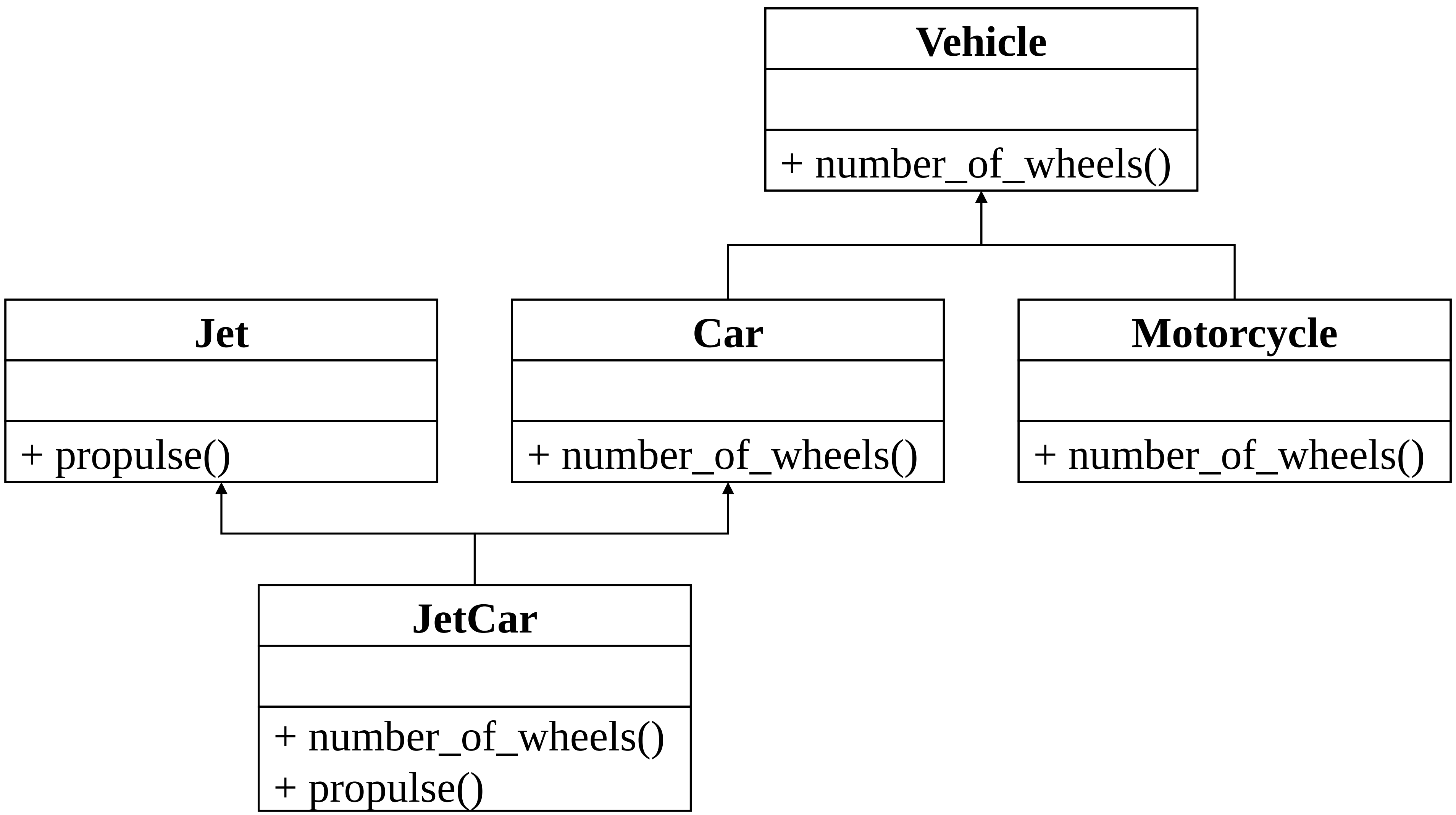
Inheritance & Polymorphism

- C++ features as **inheritance and polymorphism** make static analysis difficult to implement.
  - *multiple inheritance* in C++ includes repeated and shared inheritance of base classes, object identity distinction, dynamic dispatch that raise interesting challenges for model checking



- ESBMC replicates the methods and attributes of the base classes to the inherited class to have direct access to them
  - replicated inheritance
  - shared inheritance

Inheritance & Polymorphism



Inheritance &

```
class Vehicle
{
public:
  Vehicle() {};
  virtual int number_of_wheels() = 0;
};

class Motorcycle : public Vehicle
{
public:
  Motorcycle() : Vehicle() {};
  virtual int number_of_wheels() { return 2; };
};

class Car : public Vehicle
{
public:
  Car() : Vehicle() {};
  virtual int number_of_wheels() { return 4; };
};

int main()
{
  bool foo = nondet();

  Vehicle* v;
  if(foo)
    v = new Motorcycle();
  else
    v = new Car();

  bool res;
  if(foo)
    res = (v->number_of_wheels() == 2);
  else
    res = (v->number_of_wheels() == 4);
  assert(res);
  return 0;
}
```

Inheritance &

```cpp
class Vehicle
{
public:
  Vehicle() {};
  virtual int number_of_wheels() = 0;
};

class Motorcycle : public Vehicle
{
public:
  Motorcycle() : Vehicle() {};
  virtual int number_of_wheels() { return 2; };
};

class Car : public Vehicle
{
public:
  Car() : Vehicle() {};
  virtual int number_of_wheels() { return 4; };
};

int main()
{
  bool foo = nondet();

  Vehicle* v;
  if(foo)
    v = new Motorcycle();
  else
    v = new Car();

  bool res;
  if(foo)
    res = (v->number_of_wheels() == 2);
  else
    res = (v->number_of_wheels() == 4);
  assert(res);
  return 0;
}
```

Inheritar

```
main() (c::main):
    FUNCTION_CALL:  return_value_nondet$1=nondet()
    bool foo;
    foo = return_value_nondet$1;

    class Vehicle * v;
    IF !foo THEN GOTO 1
    new_value1 = new class Motorcycle;
    new_value1->vtable->number_of_wheels =
      &Vehicle::number_of_wheel();
    new_value1->vtable->number_of_wheels =
      &Motorcycle::number_of_wheel();
    v = (class Vehicle *)new_value;
    GOTO 2
1:  new_value2 = new class Car;
    new_value2->vtable->number_of_wheels =
      &Vehicle::number_of_wheel();
    new_value2->vtable->number_of_wheels =
      &Car::number_of_wheel();
    v = (class Vehicle *)new_value;
    bool res;
2:  IF !foo THEN GOTO 3
    FUNCTION_CALL:  return_value_number_of_wheels =
      *v->vtable->number_of_wheel()
    res = wheels == 2
    GOTO 4
3:  FUNCTION_CALL:  return_value_number_of_wheels =
      *v->vtable->number_of_wheel()
    res = wheels == 4
4:  ASERT res
    RETURN: 0
END_FUNCTION
```

Inheritan

```
return_value_nondet1 == nondet_symbol(symex::0)
foo1 == return_value_nondet1
new_value11 == new_value10
  WITH [vtable = new_value10.vtable
    WITH [number_of_wheel =
      &Motorcycle::number_of_wheels()]]
new_value12 == new_value11
  WITH [vtable = new_value11.vtable
    WITH [number_of_wheel =
      &Motorcycle::number_of_wheels()]]
v1 == new_value12
new_value21 == new_value20
  WITH [vtable = new_value20.vtable
    WITH [number_of_wheel =
      &Motorcycle::number_of_wheels()]]
new_value22 == new_value21
  WITH [vtable = new_value21.vtable
    WITH [number_of_wheel =
      &Motorcycle::number_of_wheels()]]
v2 == new_value22
v3 == (foo1 ? v1 : v2);
return_value_number_of_wheels1 == 2
res1 == (return_value_number_of_wheels1 = 2)
return_value_number_of_wheels2 == 4
res2 == (return_value_number_of_wheels2 = 4)
res3 == (foo1 ? res1 : res2)
```

Inheritance & Polymorphism

$$C := \left[\begin{array}{l} return\_value\_number\_of\_wheels_1 = 2 \\ \wedge\, res_1 = (return\_value\_number\_of\_wheels_1 = 2) \\ \wedge\, return\_value\_number\_of\_wheels_2 = 4 \\ \wedge\, res_2 = (return\_value\_number\_of\_wheels_2 = 4) \\ \wedge\, res_3 = ite(foo1, res_1, res_2) \end{array}\right]$$

$$P := \left[\; res_3 = 1 \;\right]$$

SMT-based Bounded Model Checking C++ of Programs

Encoding essential features of C++ into SMT:

(i) *Primary template and explicit-template & partial-template specialization**

(ii) Standard Template Libraries

Sequential and Associative Containers

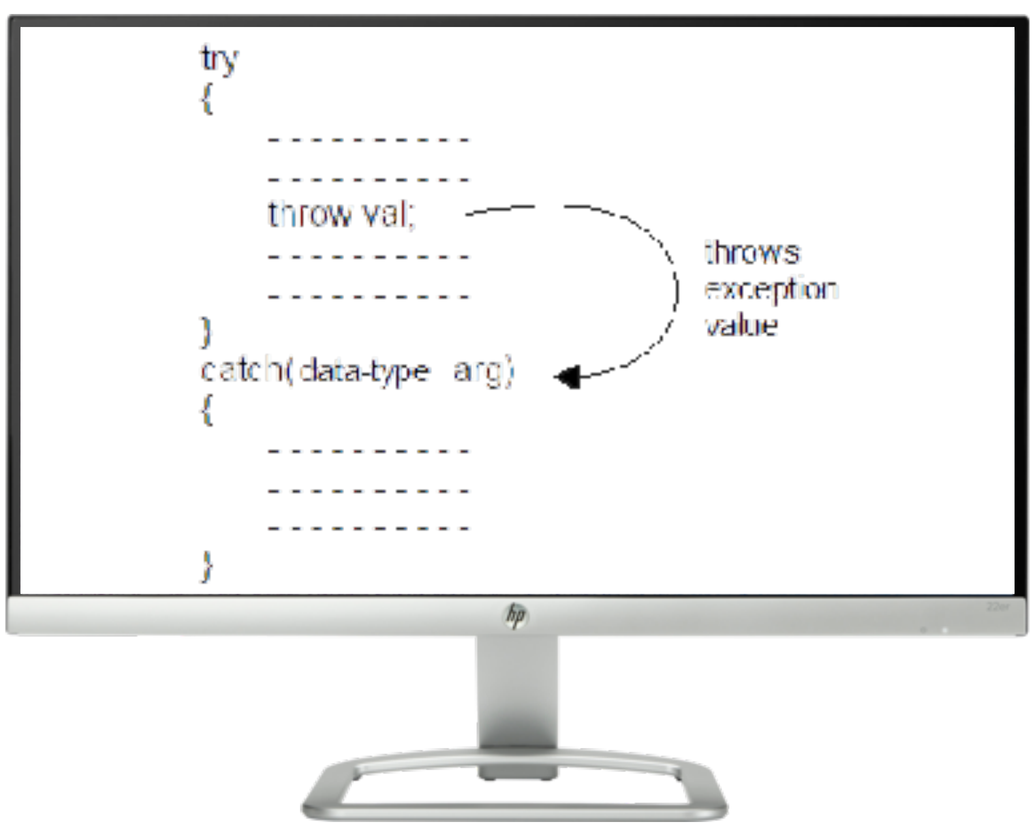
(iii) Inheritance & Polymorphism

(iv) *Exception Handling**

* *(R. Gadelha et al.)*

Try & Catch Rules

- Exceptions are **unexpected circumstances** that arise during the execution of a program, e.g., runtime errors.
  - a **try block**, where a thrown exception can be directed to a catch statement;
  - a set of **catch statements**, where a thrown exception can be handled;



- a **throw statement** that raises an exception.

Try & Catch Rules

Rule	Behavior	Formalization
r_1	Catches an exception if the type of the thrown exception e is equal to the type of the `catch` h.	$ite(\exists h \cdot M(e, h), h_{r_1} = h, h_{r_1} = h_{null})$
r_2	Catches an exception if the type of the thrown exception e is equal to the type of the `catch` h, ignoring the qualifiers *const*, *volatile*, and *restrict*.	$ite(\exists h \cdot M(e, \zeta(h)), h_{r_2} = h, h_{r_2} = h_{null})$
r_3	Catches an exception if its type is a pointer of a given type x and the type of the thrown exception is an array of the same type x.	$ite(\exists h \cdot e = e_{[]} \wedge h = h_* \wedge M(e_{[]}, h_*), h_{r_3} = h_*, h_{r_3} = h_{null})$
r_4	Catches an exception if its type is a pointer to function that returns a given type x and the type of the thrown exception is a function that returns the same type x.	$ite(\exists h \cdot e = e_{f()} \wedge h = h_{f()} \wedge M(e_{f()}, h_{f()}), h_{r_4} = h_{f()}, h_{r_4} = h_{null})$
r_5	Catches an exception if its type is an unambiguous base type for the type of the thrown exception.	$ite(\exists h \cdot U(e, h), h_{r_5} = h, h_{r_5} = h_{null})$
r_6	Catches an exception if the type of the thrown exception e can be converted to the type of the `catch` h, either by qualification or standard pointer conversion [50].	$ite(\exists h \cdot e = e_* \wedge h = h_* \wedge Q(e_*, h_*), h_{r_6} = h_*, h_{r_6} = h_{null})$
r_7	Catches an exception if its type is a void pointer h_v and the type of the thrown exception e is a pointer of any given type.	$ite(\exists h \cdot e = e_* \wedge h = h_v, h_{r_7} = h_v, h_{r_7} = h_{null})$
r_8	Catches any thrown exception if its type is ellipsis.	$ite(\forall e \cdot \exists h \cdot h = h_{\ldots}, h_{r_8} = h_{\ldots}, h_{r_8} = h_{null})$
r_9	If the `throw` expression does not throw anything, it should re-throw the last thrown exception e_{-1}, if it exists.	$ite(e = e_{null} \wedge e_{-1} \neq e_{null},$ $h'_{r_1} = r_1(e_{-1}, h_1, ..., h_n)$ $\wedge \ldots$ $\wedge\ h'_{r_8} = r_8(e_{-1}, h_1, \ldots, h_n),$ $h_{r_9} = h_{null})$

Experimental Evaluation

Evaluate accuracy & performance of model checkers targeting C++

Objectives

- Experiments aimed at answering two questions regarding correctness and performance of ESBMC:

  (EQ-I) How accurate is ESBMC when verifying the chosen C++03 programs?

  (EQ-II) How does ESBMC performance compare to other existing model checkers?

Benchmarks

- Our set of benchmarks contains **1513 C++ programs** (89,147 LOC).
  - **36% larger** than our previous published evaluation;

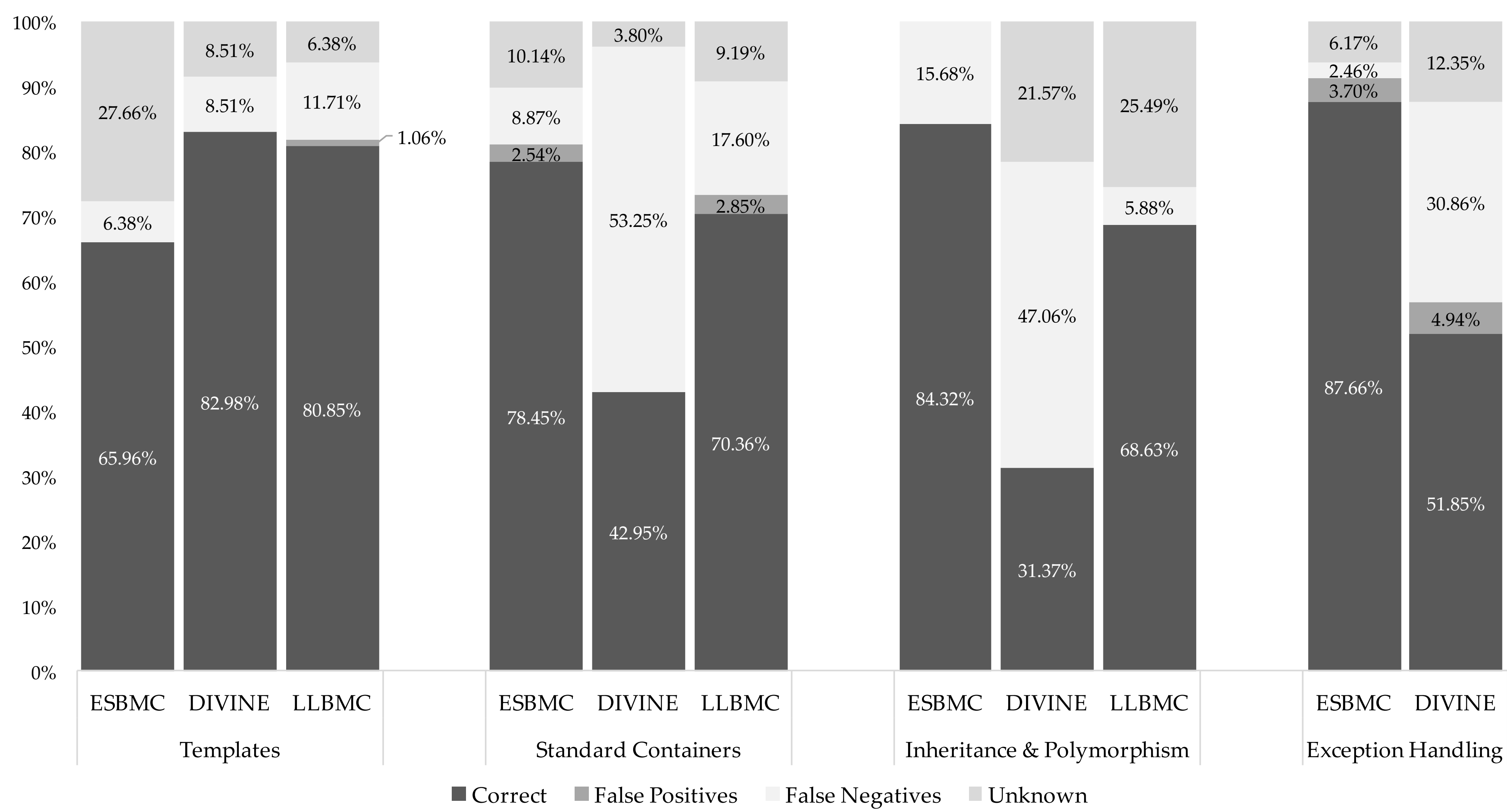
- The mentioned benchmarks are split into 5 categories:
  - **Templates**: formed by the *cbmc*, *gcc-templates* and *templates* benchmark suites (94 benchmarks);
  - **Standard Containers**: formed by *algorithm*, *deque*, *vector*, *list*, *queue*, *priority_queue*, *stack*, *map*, *multimap*, *set* and *multiset* test suites (631 benchmarks);
  - **Inheritance & Polymorphism**: formed by *inheritance* benchmark suite (51 benchmarks);
  - **Exception**: formed by the *try_catch* benchmark suite (81 benchmarks);
  - **C++03**: formed by *cpp*, *string*, and *stream* benchmark suites (656 benchmarks);

Setup

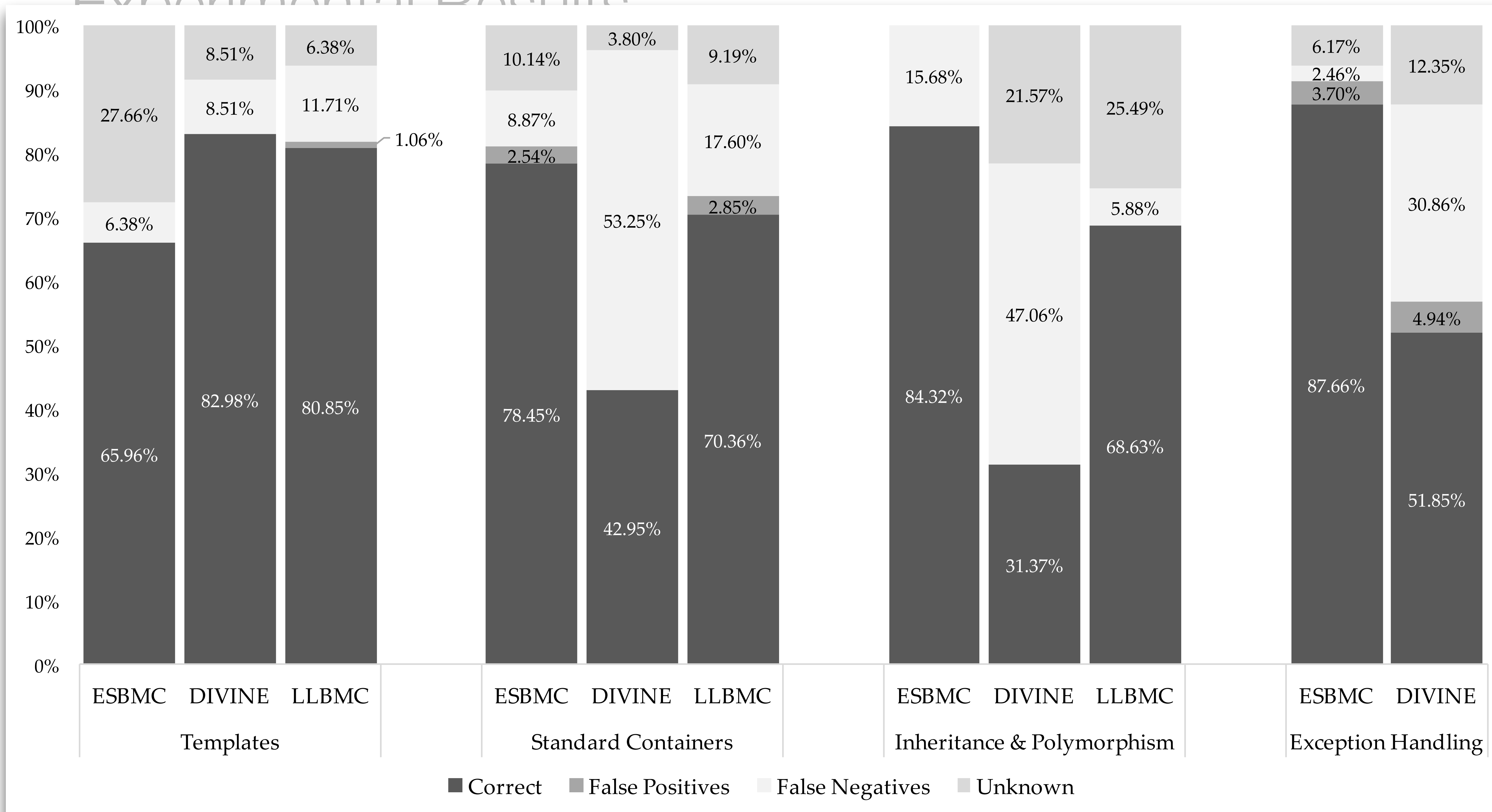
- Compare ESBMC against LLBMC and DIVINE with respect to coverage and precision in the verification process of C++03 programs
  - ESBMC *v2.0*
  - LLBMC *v2013.1*
  - DIVINE *v4.0.22*
- All experiments were conducted os
  - i7-4790 processor, 3.60GHz clock, with 16GB RAM memory
  - Ubuntu 14.04 64-bit OS
  - time limit of 900 seconds (i.e., CPU time)
  - memory limit of 14GG

Experimental Results

- A **comparison** regarding the **performance** of **LLBMC** and **ESBMC**, which are SMT-based BMC model checkers, and **DIVINE**, which employs explicit-state model checking, was carried out
  - ESBMC presented a successful rate of 85% (in **7 hours**) and LLBMC 63% (in **12 hours**), overcoming DIVINE that presented 42% (in **49 hours**)

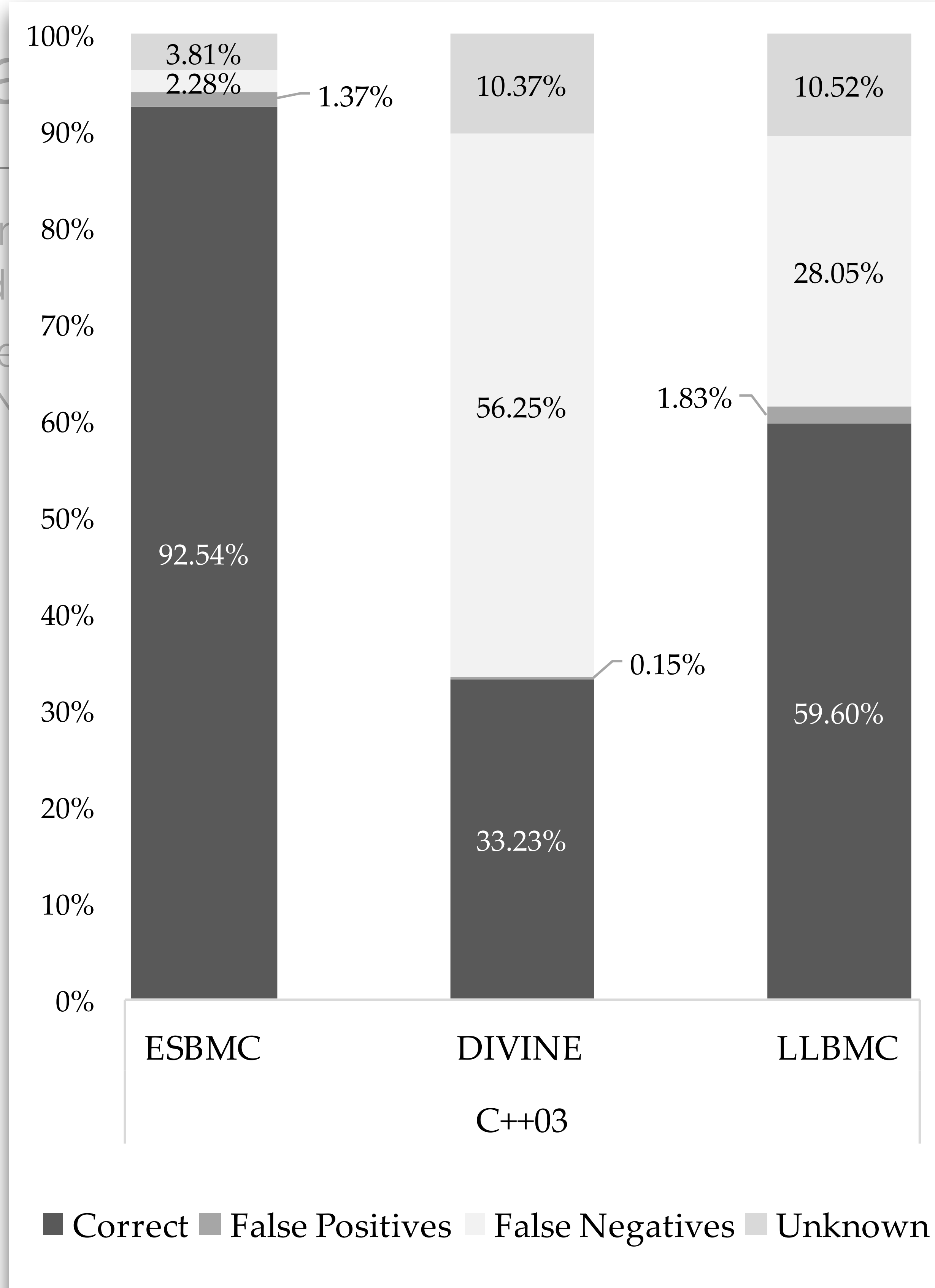


Experimental Results



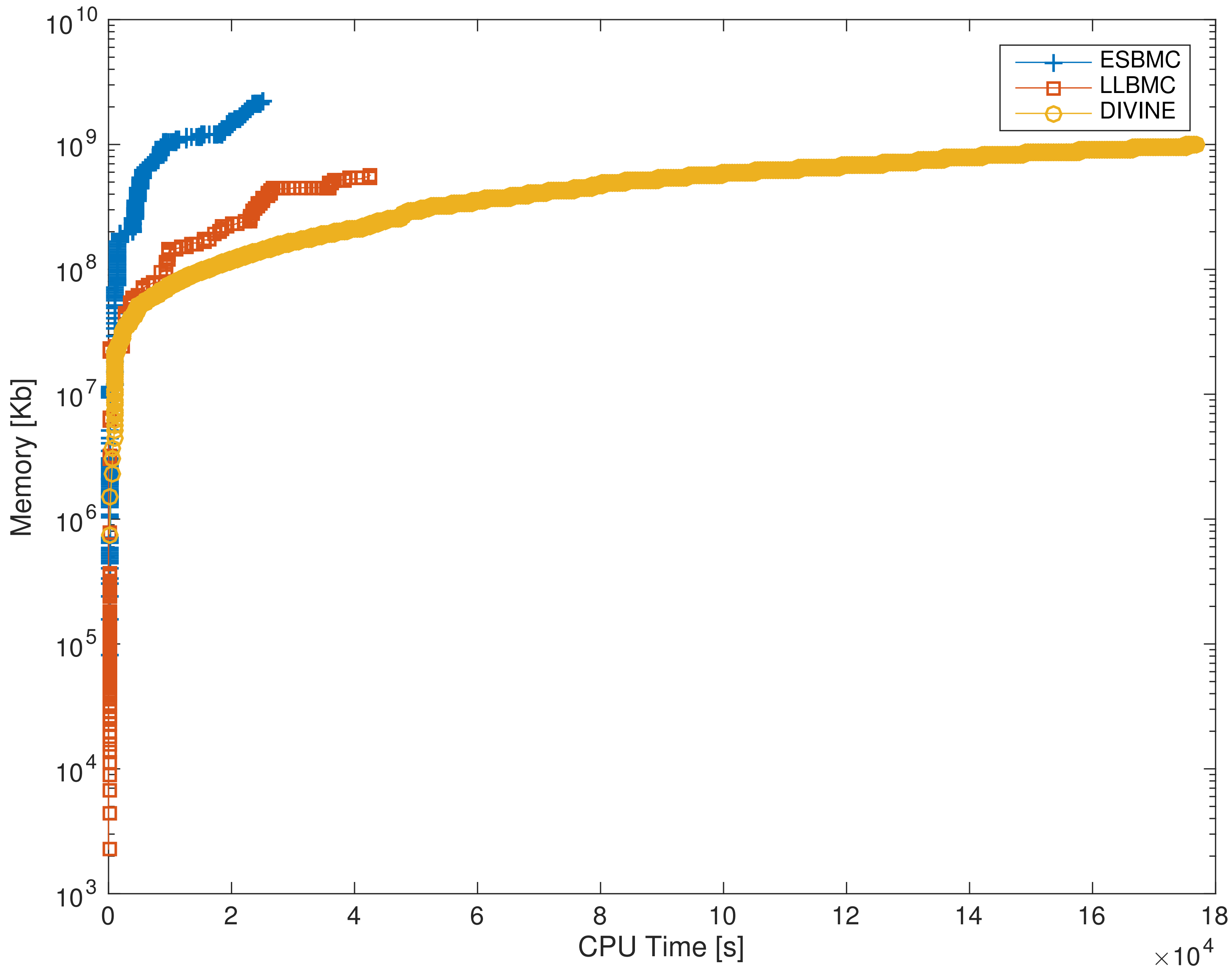
Experimenta

- A **comparison** regar… which are SMT-based BMC model checkers, and … king, was carried out
  - ESBMC presente… LBMC 63% (in **12 hours**), overcoming DIVI…



Experimental Results

- A **comparison** regarding the **performance** of **LLBMC** and **ESBMC**, which are SMT-based BMC model checkers, and **DIVINE**, which employs explicit-state model checking, was carried out
  - ESBMC presented a successful rate of 85% (in **7 hours**) and LLBMC 63% (in **12 hours**), overcoming DIVINE that presented 42% (in **49 hours**)



Conclusions

Conclusions

- This work presented an SMT-based BMC approach to verify C++03 programs using **ESBMC v2.0**

- ESBMC is able to verify correctly **84.66%** (1281 benchmarks) in 25251 seconds (approximately **7 hours**), outperforming other state-of-art C++ verification tools
    - **43.29%** and **22.27%** higher than DIVINE and LLBMC, respectively
    - **7** and **1.7** times faster than DIVINE and LLBMC, respectively

Conclusions

i. the formal description of how ESBMC handles **primary template**, **explicit-template specialization**, and **partial-template specialization**;

ii. the **operational model** structure to handle new features from the **SCL** (e.g., sequential and associative template-based containers);

iii. the formalization of the ESBMC's engine to handle **inheritance & polymorphism**;

iv. the formalization of all **throw & catch exception rules** supported by ESBMC;

v. the expressive **set of publicly available benchmarks** designed specifically to evaluate software verifiers that target the C++ programming language;

vi. the extensive comparative evaluation of **state-of-the-art software model checkers** on the verification of C++ programs;

Future Work

Although our C++ frontend is able to support most features of C++, to improve the frontend for newer versions of the C++ standard is unmanageable. Thus, one future direction is to ***rewrite ESBMC's frontend using clang to generate the program AST for C++ programs***

Future Work

Although our C++ frontend is able to support most features of C++, to improve the frontend for newer versions of the C++ standard is unmanageable. Thus, one future direction is to ***rewrite ESBMC's frontend using clang to generate the program AST for C++ programs***

Rewrite the ESBMC's frontend for C++ programs requires a major engineering effort

Future Work

Although our C++ frontend is able to support most features of C++, to improve the frontend for newer versions of the C++ standard is unmanageable. Thus, one future direction is to ***rewrite ESBMC's frontend using clang to generate the program AST for C++ programs***

- One might focus first on object-oriented aspects to set the foundation of this approach:
  - basic structures of object-oriented programs (e.g., classes, methods, constructors and destructors)
  - template instantiation
  - inheritance and polymorphism*****

Future Work

Although our C++ frontend is able to support most features of C++, to improve the frontend for newer versions of the C++ standard is unmanageable. Thus, one future direction is to ***rewrite ESBMC's frontend using clang to generate the program AST for C++ programs***

- One might focus first on object-oriented aspects to set the foundation of this approach:
    - basic structures of object-oriented programs (e.g., classes, methods, constructors and destructors)
    - template instantiation
    - inheritance and polymorphism*****

This work will set a strong foundation for the full support of C++ programming language in ESBMC.

Publications

C++

[ASE 2018] Bounded Model Checking of C++ Programs based on the Qt Cross-Platform Framework (Journal-First Abstract).

[IEEE Access 2020] *Model Checking C++ Programs.*

ESBMC

[SCP 2018] ESBMC-GPU - A Context-Bounded Model Checking Tool to Verify CUDA Programs.

[ASE 2018] ESBMC 5.0 - An Industrial-Strength C Model Checker.

[FSE 2018] Towards Counterexample-Guided *k*-Induction for Fast Bug Detection.

[NFM 2020] *Beyond k-Induction - Learning from Counterexamples to Bidirectionally Explore the State Space.*

[FASE 2020] *Scalable and Precise Verification based on the Floating-Point Theory.*

Continuous Formal Verification

[TAPAS 2019] Continuous Formal Verification at Scale.

[ICSE 2020] Code-Level Model Checking in the Software Development Workflow.

Media

[Eldorado Institute] Verificação Formal e seu Papel no Desenvolvimento de Sistemas Cyber-Físicos Críticos.

Publications

C++

[ASE 2018] Bounded Model Checking of C++ Programs based on the Qt Cross-Platform Framework (Journal-First Abstract).

[IEEE Access 2020] *Model Checking C++ Programs.*

Under review

ESBMC

[SCP 2018] ESBMC-GPU - A Context-Bounded Model Checking Tool to Verify CUDA Programs.

[ASE 2018] ESBMC 5.0 - An Industrial-Strength C Model Checker.

[FSE 2018] Towards Counterexample-Guided *k*-Induction for Fast Bug Detection.

[NFM 2020] *Beyond k-Induction - Learning from Counterexamples to Bidirectionally Explore the State Space.*

[FASE 2020] *Scalable and Precise Verification based on the Floating-Point Theory.*

Under review

Under review

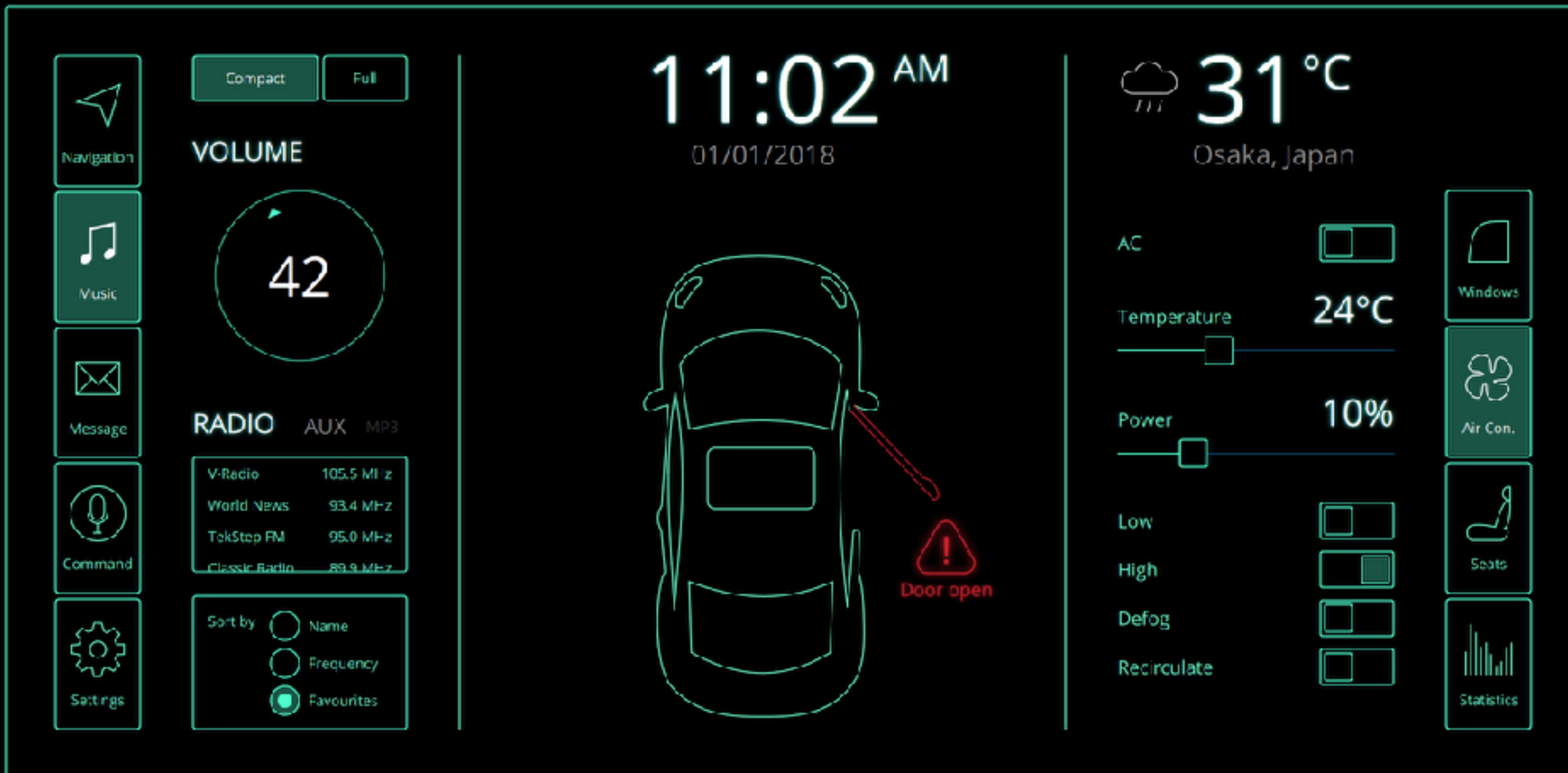
Continuous Formal Verification

[TAPAS 2019] Continuous Formal Verification at Scale.

[ICSE 2020] Code-Level Model Checking in the Software Development Workflow.

Media

[Eldorado Institute] Verificação Formal e seu Papel no Desenvolvimento de Sistemas Cyber-Físicos Críticos.



Formal Verification to Ensuring the Memory Safety of C++ Programs

Felipe R. Monteiro