Synthesis of Synchronization using Uninterpreted Functions*

October 22, 2014

Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Außerlechner, and Raphael Spörk

* This work was supported in part by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23) and the project QUAIN (I774-N23).
What is Synthesis?

- **Specification:** What?
  - **From:** Graz, Inffeldgasse
  - **To:** Lausanne, 6pm

- **Implementation:** How?
  - Walk to Moserhofgasse
  - Tram 6 to Jakominiplatz
    - Buy tram ticket
  - Tram 3 to train station Graz
  - Buy train ticket
  - Train to Salzburg
  - Train to Zürich
  - Train to Lausanne
  - Walk to Lausanne Fon
  - And so on …
What is Synthesis?

- **Specification:** What?
  - **From:** Graz, Inffeldgasse
  - **To:** Lausanne, 6pm

- **Implementation:** How?
  - Walk to Moserhofgasse
  - Tram ??? to Jakominiplatz
    - Buy tram ticket
  - Tram 3 to train station Graz
  - Buy train ticket
  - Train to ???
  - Train to Zürich
  - Train to Lausanne
  - Walk to Lausanne Fon
  - And so on …
Concurrent Programs

Functionality:
- Hard to specify
- Easy to implement
  \[\rightarrow\text{Implement manually}\]

Concurrent Correctness:
- Easy to specify
  - Same result
- Hard to implement
  \[\rightarrow\text{Synthesize}\]

Vision:

Sequentially Correct Code \[\rightarrow\text{Same Results}\] Synthesizing Compiler \[\rightarrow\text{Parallel Code}\]
Synthesizing Atomic Sections

Example:

- **RSA decryption using Chinese Remainder Theorem**
  - **Goal:** \( m = c^d \mod (p \times q) \)
  - **Faster:** \( m_p = c^d \mod p \quad m_q = c^d \mod q \quad m = \text{crt}(m_p, m_q) \)

**Parallelization:**

```plaintext
thread1() {
  m_p := c^d \mod p;
  fin_1 := true;
  if(!merged && \&\& fin_2)
    merged := true;
  m_p := \text{crt}(m_p, m_q);
}

thread2() {
  m_q := c^d \mod q;
  fin_2 := true;
  if(!merged && \&\& fin_1)
    merged := true;
  m_p := \text{crt}(m_p, m_q);
}
```
Flow

Program → Abstraction → Atomic Sections → SMT Encoding → Verification → Synchronized Program

→

Counterexample Analysis

Verification

Synchronized Program

Abstraction

SMT Encoding

Atomic Sections
Abstraction

Challenge: Complicated arithmetic

- Synchronization should not depend on arithmetic
- \( \rightarrow \) Abstract using uninterpreted functions

```
thread1() {
    m_p := c^d \mod p;
    fin_1 := true;
    if(!merged && fin_2)
        merged := true;
    m_p := \text{crt}(m_p, m_q);
}
```

```
thread2() {
    m_q := c^d \mod q;
    fin_2 := true;
    if(!merged && fin_1)
        merged := true;
    m_p := \text{crt}(m_p, m_q);
}
```
Challenge: Complicated arithmetic

- Synchronization should not depend on arithmetic
- → Abstract using uninterpreted functions
  - All arithmetic operations: +, -, *, ...
  - Calls of functions without side-effects

```
thread1() {
  m_p := f_me(c, d, p);
  fin_1 := true;
  if(!merged && fin_2)
    merged := true;
  m_p := f_crt(m_p, m_q);
}

thread2() {
  m_q := f_me(c, d, q);
  fin_2 := true;
  if(!merged && fin_1)
    merged := true;
  m_p := f_crt(m_p, m_q);
}
```
Flow

Program → Abstraction → SMT Encoding → Verification → Synchronized Program

Atomic Sections → Counterexample Analysis

Robert Könighofer
Synthesis of Synchronization using Uninterpreted Functions
SMT Encoding

- Implicit specification
  - \(\text{result}(\text{Thread1} \ || \ \text{Thread2}) = \text{result}(\text{Thread1} \circ \text{Thread2}) \lor \text{result}(\text{Thread2} \circ \text{Thread1})\)
  - \(\text{result}()\): global variables at termination
  - Often called “serializability” or “linearizability”

- Construct SMT formula:
  - \(\text{incorrect}(\text{inputs, scheduling})\)
  - Satisfying assignment = incorrect execution

- Approach based on Bounded Model Checking [CAV’05]
  - Loops are unrolled
  - Function calls are inlined (or abstracted)
Flow

Program → Abstraction → SMT Encoding → Verification → Synchronized Program

Atomic Sections → Counterexample Analysis → UNSAT counterexample

SMT Solver
Counterexample Analysis: Method 1 [POPL’10]

- Eliminate counterexample:
  - Atomic section at $A \lor B$
Counterexample Analysis:
Method 1 [POPL’10]

- Eliminate counterexample:
  - Atomic section at $A \lor B$
  - Atomic section at $A \lor D$

Iteration 2:

Thread 1
- Line A
- Line C (end of T1)

Thread 2
- Line D
Counterexample Analysis: Method 1 [POPL’10]

Iteration 3:

- Eliminate counterexample:
  - Atomic section at $A \lor B$
  - Atomic section at $A \lor D$
  - Minimal satisfying assignment
  - $\Rightarrow$ Atomic section at $A$

No more counterexamples
Counterexample Analysis: Method 2

- Start with last (non-mandatory) thread switch \( B \)
- Can we build a valid run from \( B \) on?
Counterexample Analysis: Method 2

- Start with last (non-mandatory) thread switch \( B \)
- Can we build a valid run from \( B \) on?
  - No? Problem already before
    - Investigate \( A \) in the same way
  - Yes? \( B \) is suspicious.
    - Add atomic section at \( B \)

- This is a heuristic!
  - May not find the minimal solution
Flow

Program → Abstraction → SMT Encoding → Verification → Synchronized Program

Atomic Sections → Counterexample Analysis

SMT Solver

counterexample

UNSAT
Experimental Results

- Prototype tool for (simple) C programs
- Toy examples:
  - **linEq:**
    - Given: linear equation \(4a + 3b + 9c - 4d = 6\)
    - Given: assignment \(a=100, b=0, c=3, d=12\)
    - Program performs parallelized check
    - Abstraction: \(+, \ast \rightarrow f_+, f_\ast()\)
  - **VecPrime:**
    - Counts prime numbers in a vector
    - Abstraction: \(\text{isPrime}() \rightarrow f_p()\)
Experimental Results: Toy Examples
Speedup due to Abstraction

<table>
<thead>
<tr>
<th>Method</th>
<th>With abstraction (UIF) [sec]</th>
<th>Without abstraction [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method 1</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Method 2</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Average speedup factor:
- 110 not counting time-outs
- 160 when counting time-outs
Experimental Results

- Real-world examples:
  - CVE-2014-0196 bug in Linux TTY driver
    - Race condition can produce buffer overflow
Synthesis of Synchronization using Uninterpreted Functions

Real-world examples:
- CVE-2014-0196 bug in Linux TTY driver
- Race condition can produce buffer overflow

```c
int tty_size;
int tty_offset;
int OPOST_tty;
int STATE = 1;
void thread1() {
    int c = 0;
    int nr = 22;
    int b = 77;
    int true_int = 1;
    while(true_int == 1) {
        if(OPOST_tty) {
            STATE = 2;
            while(nr > 0) {
                int num = nr + 3;
                b = b + num;
                nr = nr - num;
                if(nr != 0) {
                    c = b;
                    b = b + 1;
                    nr = nr - 1;
                }
            }
        } else {
            STATE = 3;
            while(nr > 0) {
                int tmpOffset = tty_offset;
                int tty_space_left = tty_size - tmpOffset;
                if(tty_space_left - nr >= 0) {
                    c = nr;
                } else {
                    c = tty_space_left;
                    tmpOffset = tty_offset;
                    tmpOffset = tmpOffset + c;
                    tty_offset = tmpOffset;
                    if(c > 0) {
                        b = b + c;
                        nr = nr - c;
                    }
                }
            }
        }
    }
}
```
Experimental Results

- Real-world examples:
  - CVE-2014-0196 bug in Linux TTY driver
    - Race condition can produce buffer overflow
  - Race condition in iio-subsystem of linux-kernel
    - Variable that counts the number of running threads
  - Race condition in broadcom tigon3 ethernet driver
    - Statistics can get inconsistent
Experimental Results: Real-World Bugs

- **TTY and Tigon3:**
  - Our tool finds exactly the suggested fix
- **IIO:**
  - Our tool finds a slightly different fix
- **No user-defined specification necessary**
  - Serialzability as implicit specification is enough
- **Execution times [sec]:**

<table>
<thead>
<tr>
<th></th>
<th>Without Abstraction</th>
<th>With Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method 1</td>
<td>Method 2</td>
</tr>
<tr>
<td>TTY</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>IIO</td>
<td>1.1</td>
<td>1.3</td>
</tr>
<tr>
<td>Tigon3</td>
<td>17</td>
<td>21</td>
</tr>
</tbody>
</table>
Summary and Conclusions

Highlights:
- No manual specifications $\rightarrow$ usability
- Abstraction with uninterpreted functions $\rightarrow$ scalability
- Proof-of-concept implementation
  - [http://www.iaik.tugraz.at/content/research/design_verification/atoss/](http://www.iaik.tugraz.at/content/research/design_verification/atoss/)

Future work:
- Abstraction refinement (e.g., associativity, commutativity), other abstractions, loops, …
References
