Correct-by-Construction Code Synthesis from Formal Models for Safe and Secure Applications

Topic Area: Usability of Formal Methods towards System Certification

Sandeep Shukla
FERMAT Lab,
Centre for Embedded Systems for Critical Applications
Bradley Department of Electrical and Computer Engineering
Virginia Tech, USA.

shukla@vt.edu

This research was funded by Air Force Rome Labs
Safety-Critical Software
Safety-Critical Software Requirements

- Deterministic
- Functionally Correct
- Tolerant to Fault Models
- Satisfies Real-Time Properties
Traditional Development

A posteriori verification – cost, delays, quality problems

Clarus Concept of Operations, Publication No. FHWA-JPO-05-072, Federal Highway Administration (FHWA), 2005
A Priori Verification and Verified Refinement

- Requirements models << Implementation
- Implementation language may not have formal semantics
- Model extraction may induce abstraction error

We Propose: Verify at high abstraction level → Synthesize implementation by correctness preserving refinement steps
Correct-by-Construction Synthesis

• A Modeling Language
  – needed to capture intended computation and communications
  – Must be at a high enough abstraction level
  – must have formal semantics
  – must be verifiable using formal techniques
  – refinements towards implementation must be provably correctness preserving steps
All possible Behaviors of a Program could have millions of paths

Testing, Coverage (MC/DC and others) cannot guarantee too much
Modeling Language candidates

State based: Automata

State based: Kripke Structure

Temporal Logic Based

$\text{AG } p$

$\text{EF } q$
Requirements Models

• These usually
  – Are highest level
  – Abstracts away implementation constraints
  – Abstract behavior may be nondeterministic
  – All behaviors possible in the requirements model may not be implemented based on additional constraints
  – Correctness of implementation = set of behaviors of the spec is a super set of behaviors of the implementation
What level of Abstraction is Given?

- What if control engineer already gave me an algorithm for control?
- What if O/S designer already has a detailed algorithm in a pseudo notation
- What if the data path is already given
- Then implementation behavior must match the abstract behavior
- Some cases, it can be a subset
Stream based Specification

• Embedded systems are long lived
• React to input events
• The inputs keep coming – hence form a set of streams
• During processing we convert one stream to another and so on until outputs are computed
Input Data/Event Streams get processed, come out as Output Data/Event Streams. Sometimes, feedback is used to retain state information.
Choices

- Control/Dataflow Automata
- Parallel Control/Dataflow Automata
- Dataflow networks
  - Synchronous Data flow (SDF)
  - ..... 
  - KPN
- Petrinets
- Process Algebras
- .....
Dataflow Network

• Determinism at crucial locations of computation loci – important
• Local non-determinism – fine
• Composability – important
  – KPN – too strong requirements
  – SDF – too low expressibility
  – Single Clocked dataflow vs. Multi-clocked dataflow
Data flow examples

• \( y := f(x) + g(x) \mid o := h(y) \)

Input x
Output o
Internal y

While (1){
    Read(x);
    y = f(x) + g(x);
    o = h(y);
}

NO STATE VARIABLE
Example 2

- $S := S' + x \mid S' = S$ \init 0$

Input $x$;
Output $S$;
Internal $S'$;
$S' = 0$;
While (1) {
- Read($x$);
- $S = S' + x$;
- Output($S$);
- $S' = S$;
}
Example 3

\[ y := x \text{ when } c \mid c := \text{true when } (x > 0) \text{ default false} \]

```
input x;
output y;
internal c;
c = false;
while(1){
    read(x);
c = (x > 0) ? T : F;
    if (c)
        y = x;
}
```
Example 4

- \( y := (x > 0) \) | \( z := ((x + 1) \text{ when } y) \text{ default } x \)

```
input x;
output z;
Internal y;
y = false;
While(1){
  Read(x);
  Y = (x>0);
  if (y) {
    z = x + 1;
  } else
    z = x;
}
```
MRICDF Dataflow Network
Model of Time

**Events:** Each input, output or internal variable changes value by computation or external intervention – each such occurrence

**Ordering:** Events are temporally and spatially ordered. Temporally they may be partially ordered (think threads)

**Signal:** Events occurring to a single variable (input, output, or internal) have to be totally ordered

**Reaction:** Computation triggered by one or more input events, may or may not require other inputs during the reaction, a reaction ends by computing all the required output events

**Ordering of Reactions:** Reactions are maximal computation in response to one or more input events, leading to internal variable, output computation, and cascading in response to those changes – until no more new event can occur without another stimulus event
Model of Time (2)

**Logical Time:** Each reaction is called an abstract instant or logical instant

**Ordering of Logical Instants:** Partially ordered but may be sequential

**Synthesis of Dataflow Specifications:** creating equivalence classes of reactions that are characterized by same data flow

Not all logical instants has the same data flow – an input may be present in one logical instant, and in another – not

Once the equivalence classes of reactions are found – schedule the reactions

Then order the equivalence classes in the right implementation code
What is MRICDF?

• Multi-Rate Instantaneous Communication Data Flow
• A Visual Language (with a textual substitute) to express a computation over concurrent streams of data
• A stream of data/events → a totally ordered set of events
• Why care about streams of Data?
How many equivalence classes?

One.

One reaction type – keeps repeating sequentially ad infinitum.

Read(x);
y = f(x) + g(x);
o = h(y);
Example 2

How many equivalence classes?

One.

One reaction type – keeps repeating sequentially ad infinitum.

Read(x);
S = S' + x;
Output(S);
S' = S;
Example 3

How many equivalence classes? Read(x);

When $x > 0$

- $c = \text{true}$;
- $y = x$;

When $x \leq 0$

- $c = \text{false}$;
Example 3.1

How many equivalence classes?

READ(x)

when $x > 0$

$c = \text{true}$

$y = x$

$z = T$

when $x \leq 0$

$c = \text{false}$

$y = x$

$z = F$

$y = x$

$z = F$

$z = F$
Input $x$, $y$

Output $u$, $v$, $z$

Internal $w$, $p$

$u = f(x)$ |
$v = g(y)$ |
$w = u$ when $(x > 5)$ |
$p = v$ when $(y > 5)$ |
$z = h(w, p)$

Additional constraint

$(x > 5) \ ^\land \ (y > 5)$
Synchronization = Stretch an Instant

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>u</th>
<th>v</th>
<th>w</th>
<th>p</th>
<th>z</th>
<th>x &gt; 5</th>
<th>y &gt; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>f(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>g(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>f(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>f(6)</td>
<td>g(1)</td>
<td>f(6)</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>g(9)</td>
<td></td>
<td>g(9)</td>
<td>h(f(6), g(9))</td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>f(8)</td>
<td></td>
<td>f(8)</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>f(1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>
Example 4 requires multithreading

Thread 1

while (1){
    b = false;
    Read(x);
    u = f(x);
    If (x> 5){
        b = true;
        wait(e1);
    }
}

Thread 2

while (1){
    c = false;
    Read(y);
    v = g(y);
    If (y > 5) {
        c = true;
        wait(e2);
    }
}

Thread 3

while (1){
    while (b != true || c != true) ;
    z = h(u,v);
    notify(e1,e2);
}

u

v

b

c

}
Finding those Reaction Equivalence Classes

• Clock Calculus
  – Create the order in which inputs must be read or outputs must be computed or internal variables must be computed
  – Each leaf of the resulting tree will provide the different possible equivalence classes of reactions
  – Within each leaf the computation is ordered based on data dependencies
    • Must check there is not cyclic dependencies (causality)
  – The over all schedule is read off the clock tree, and the leafs
Algorithm for Clock Calculus

• Create a Boolean Abstraction
• Resulting theory’s prime implicate is computed and propagated recursively

\[ y := \text{x when } c \mid c := \text{true when } (x > 0) \]
default false

\[
\begin{align*}
\text{b}_y &= \text{b}_x \land \text{b}_c \\
\text{b}_c &= \text{b}_{[x>0]} \lor \text{false} \\
\text{b}_c &= \text{b}_{[c]} \lor \text{b}_{[-c]} \\
\text{b}_{[c]} \land \text{b}_{[-c]} &= 0 \\
\text{b}_x &= \text{b}_{[x>0]} \lor \text{b}_{[x \leq 0]} \\
\text{b}_{[x>0]} \land \text{b}_{[x \leq 0]} &= 0 \\
\text{b}_x \lor \text{b}_y \lor \text{b}_c
\end{align*}
\]

Unitary Prime Implicate: \( b_x \), and \( b_c \)
But \( x \) determines \( c \), so \( x \) is the main trigger.
Putting \( b_x = 1 = b_c \)

\[
\begin{align*}
\text{b}_y &= \text{b}_{[c]} \\
\text{b}_{[c]} \lor \text{b}_{[-c]} &= 1 \\
\text{b}_{[x>0]} \lor \text{b}_{[x \leq 0]} &= 1 \\
\text{b}_{[c]} \land \text{b}_{[-c]} &= 0 \\
\text{b}_{[x>0]} \land \text{b}_{[x \leq 0]} &= 0
\end{align*}
\]
FERMAT

SSL
- System Security Policies
  - Application Security
  - O/S Security
  - Middleware Security
  - Firmware Security

AADL
- System Architecture Constraints
  - Platform Constraints
    - HW Constraints
    - Bus/Network Constraints
    - Softw. Arch Constraints
  - Processor Constraints
  - Memory Constraints
  - Firmware Constraints
  - Distribution/Protocol Constraints
- Performance Constraints

SCR
- Abstract Requirements Models
- Concrete Requirements Models

MRICDF
- Concrete Control Models
- OS Functional Specs
- Middleware Functional Specs

Synthesis Engine
- Control apps synthesis
- Middleware synthesis
- OS comp. synthesis
- Firmware config. Syn.

Kernel & sys call libraries
- Device interface library
- Security libraries
- Apps libraries

Handwritten or separately synthesized

Control
Middleware
OS Comp
Firmware Config.
An Alternative Use Case
Simulink Model of Air Condition Control
MRICDF Model for the Simulink Model
Multi-core processors being common in desktop/server/laptops/ipad2 – embedded multi-core processors will also be common in the near future. Programming software for such target processors require concurrency management, deadlock, starvation freedom, safety, and real-time and fault-tolerant operation. Verification being hard, formal correct-by-construction synthesis is a way to mitigate correctness problems.

**Current State:** programming models for concurrent multi-threaded software too complex and patch work on sequential thinking. We need a new and correct-by-construction approach to programming multi-core systems.

**Military Impact/Payoff:** Most Mission Critical Systems have safety-critical embedded software – millions of lines of code --- abundant concurrency- real-time and fault-tolerant execution. Correctness is of paramount importance – post facto verification is hard. We want to solve this problem so trust worthy system can be created by construction.

This is a step towards a science of parallel programming for multi-core.

**Approach:** Polychronous modeling for formally capturing requirements – relaxation of linearizability -- weaker models of consistency – concurrent data structure, synchronization construct synthesis – execution time prediction – fault tolerance by timing and resource redundancy.

**Deliverables/Products**
1. Characterizations of specifications for synthesizability under various concurrency correctness conditions
2. Algorithmic Methods for Synthesis
3. Algorithmic methods for worst case execution time prediction, optimization and guarantees
4. Modeling faults and tolerant designs and synthesis techniques

**Metrics to Measure Success:**
Application of the algorithms and tools on a reasonable size experimental software to measure the productivity gain with respect to standard software programs for multicore -- demonstrate at least 30% productivity gain.