Verifying Cyber-Physical Systems by Combining Software Model Checking with Hybrid Systems Reachability

Stanley Bak, AFRL
Sagar Chaki, SEI/CMU

October 4, 2016

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Motivation

Cyber-Physical System (CPSs) play safety-critical roles in day-to-day lives

- Avionics, automotive, healthcare, energy

High-level of assurance of safe and secure behavior desired

- As close to the executable as possible

Formal verification provides high confidence in principle, but

- **Issue 1**: Application and controller algorithms analysed by different techniques – each with their own specialized tools
- **Issue 2**: In practice plagued by scalability issues

Can **compositional reasoning** address both issues?

We present a compositional approach to verify CPS software

- Software model checking + hybrid system reachability
- Validated on a multi-agent collision avoidance protocol
CPS Model Of Computation

System composed of application $A$ and controller $C$
- Execute concurrently: $S = A \parallel C$
- Communicate via shared variables
  - Cyber variables $V_C$ written by $A$ and read by $C$
  - Physical variables $V_P$ written by $C$ and read by $A$
- Accessed by $A$ via API functions
- Application $A$ available as source code
- Controller $C$ available as a hybrid automaton
  - $C = \text{controller} + \text{plant}$ (from control theory perspective)

Want to verify that $S$ satisfies a safety property (something bad never happens)
- Formally, $S \models \Phi$ where $\Phi$ is an invariant expressing the safety property of interest
Example: 2D Quadcopter Movement

Current setpoint $sp_{cur} = (0,0)$

Position $pos = (0,0)$

Next setpoint $sp_{nxt} = (5,0)$

Cell Ids $x \rightarrow 0 \ 1 \ 2 \ 3 \ ...$

Positions $-2.5 \ 2.5 \ 7.5 \ 12.5 \ 17.5$
Example: Target Property

\[ \Phi_{\text{hover}} \equiv |pos - spcur| \leq (1.5, 1.5) \]

\[ (\Phi_{\text{hover}} \land sp\text{next} = sp\text{cur}) \]

\[ \lor \]

\[ (\Phi_{\text{move}} \land (|sp\text{next} - sp\text{cur}| = (5,0) \lor |sp\text{next} - sp\text{cur}| = (0,5))) \]

\[ \Phi_{\text{move}} \equiv \min(sp\text{cur}_x, sp\text{next}_x) - 1.5 \leq pos_x \]

\[ \leq \max(sp\text{cur}_x, sp\text{next}_x) + 1.5 \]

\[ \land \min(sp\text{cur}_y, sp\text{next}_y) - 1.5 \leq pos_y \]

\[ \leq \max(sp\text{cur}_y, sp\text{next}_y) + 1.5 \]
Example: 2D Quadcopter Movement

Periodically invokes API functions
`update_setpoint(x, y)`
and `has_arrived()`
that update `spcur` and `spnxt` to interact with the controller.

Continuously executes a control algorithm to move/hover the platform based on values of `spcur` and `spnxt`. Updates `pos`.

**Shared Variables**
Cyber: `spcur`, `spnxt`  
Physical: `pos`  

**API Function Parameters**
`x`, `y`
Verification Approach

No existing tools to verify (source code + hybrid automata)
- But each domain has its own specialized tools: software model checkers and hybrid reachability checkers
- Developing such a tool that combines the statespace $A$ and $C$ in a brute-force way will not scale

Insight: application and controller make assumptions about each other to achieve overall safe behavior

Approach:
- Use “contract automaton” to express inter-dependency between $A$ and $C$
- Separately verify that $A$ and $C$ implement desired behavior under the assumption that the other party does so as well
- Use an “assume-guarantee” style proof rule to show the $A \parallel C \models \Phi$
Benefits of Verification Approach

Use “contract automaton” to express inter-dependency between $A$ and $C$
  • Explicit formal understanding between teams developing $A$ and $C$

Separately verify that $A$ and $C$ implement desired behavior under the assumption that the other party does so as well
  • Compositional $\Rightarrow$ more scalable
  • Use domain-specific tools $\Rightarrow$ build on progress in each area

Use an “assume-guarantee” style proof rule to show the $A \parallel C \models \Phi$
  • Proof-rule formally proven to be sound $\Rightarrow$ amortized proof cost
  • Other variants can be developed to manage tradeoff between completeness and verification complexity
Example: Assumptions between $A$ and $C$

(C1) The application always calls $update\_setpoint(x, y)$, with arguments that satisfy the condition $|(x, y) - spcur| = (5, 0) \lor |(x, y) - spcur| = (0, 5)$.

(C2) Once the application calls $update\_setpoint(x, y)$, it can keep calling $has\_arrived()$ until it gets a return value of $\text{TRUE}$; once $has\_arrived()$ returns $\text{TRUE}$, the application can only then start to call $update\_setpoint(x, y)$ again.

(C3) When the quadcopter is hovering (i.e., $spnxt = spcur$), the controller must maintain the following invariant: $\Phi_{hover} \equiv |pos - spcur| \leq (1.5, 1.5)$.

(C4) When the quadcopter is moving (i.e., $|spnxt - spcur| = (5, 0) \lor |spnxt - spcur| = (0, 5)$), the controller must maintain the following invariant:

$$\Phi_{move} \equiv min(spcur_x, spnxt_x) - 1.5 \leq pos_x$$
$$\leq max(spcur_x, spnxt_x) + 1.5$$
$$\land min(spcur_y, spnxt_y) - 1.5 \leq pos_y$$
$$\leq max(spcur_y, spnxt_y) + 1.5$$
**Example: Contract Automaton**

$C_1$ and $C_2$ are enforced by the possible transitions and the function calls labeling them.

$C_3$ and $C_4$ are enforced by the invariants labeling the locations.
Contract Automaton Invariant = Target Property

\[ \text{hover} \]

\[ spnxt = spcur \]
\[ \Phi_{\text{hover}} \]

\[ f: \text{update}_\text{setpoint}(x, y) \]
\[ \text{req: } |(x, y) - spcur| = (5,0) \]
\[ \lor |(x, y) - spcur| = (0,5) \]
\[ \text{grd: true} \]
\[ A: \langle spnxt := (x, y) \rangle \]
\[ rv: \Diamond \]

\[ f: \text{has}_\text{arrived}() \]
\[ \text{req: true} \]
\[ \text{grd: } |\text{pos} - spnxt| \leq (0.1,0.1) \]
\[ A: \langle spcur := spnxt \rangle \]
\[ rv: \text{true} \]

\[ |spnxt - spcur| = (5,0) \lor |spnxt - spcur| = (0,5) \]
\[ \Phi_{\text{move}} \]

\[ f: \text{has}_\text{arrived}() \]
\[ \text{req: true} \]
\[ \text{grd: } |\text{pos} - spnxt| > (0.1,0.1) \]
\[ A: \langle \rangle \]
\[ rv: \text{false} \]

CA Invariant = disjunction of state invariants

\[(\Phi_{\text{hover}} \land spnxt = spcur) \]
\[\lor \]
\[(\Phi_{\text{move}} \land (|spnxt - spcur| = (5,0) \lor |spnxt - spcur| = (0,5)))\]
Assume-Guarantee Proof Rule

Premise 1: Application $A$ refines the contract automaton $M$ (calls API functions in the right order and with proper arguments)

Premise 2: Controller $C$ refines the contract automaton $M$ (keeps the physical state within required bounds)

Theorem 1 (Compositional Refinement).

\[
\begin{align*}
A \preceq M & \quad C \preceq M \\
A \parallel C \preceq M
\end{align*}
\]

Conclusion: System satisfies all invariants of the contract automaton $M = \text{target safety property}$

\[
(\Phi_{\text{hover}} \land \text{spnxt} = \text{spcur}) \lor (\Phi_{\text{move}} \land \\
(|\text{spnxt} - \text{spcur}| = (5, 0) \lor |\text{spnxt} - \text{spcur}| = (0, 5)))
\]
Discharging The Premises

Premise1: Application $A$ refines the contract automaton $M$ (calls API functions in the right order and with proper arguments)
  - Reduced to software model checking, discharged via CBMC
  - Manually supplied invariants and used CBMC to verify that they are inductive
  - 1700 LOC, 2.9GHz, 16GB RAM, 3.5 seconds

Premise2: Controller $C$ refines the contract automaton $M$ (keeps the physical state within required bounds)
  - Reduced to hybrid system reachability, discharged via SpaceEX
  - Required continuous approximation and symmetry argument
  - 2.3GHz, 16GB RAM, 33 seconds
Discharging Premise 1

\[ \Phi_{\text{hover}} \]

\[ spnxt = spcur \]

\[ f: \text{update\_setpoint}(x, y) \]

\[ \text{req: } |(x, y) - spcur| = (5,0) \]
\[ \lor |(x, y) - spcur| = (0,5) \]

\[ A: \langle \text{spnxt} := (x, y) \rangle \]

\[ rv: \diamond \]

\[ |\text{spnxt} - spcur| = (5,0) \lor |\text{spnxt} - spcur| = (0,5) \]

\[ \Phi_{\text{move}} \]

\[ \text{hover} \]

\[ \text{wait} \]

```
enum Loc {hover, wait};
Loc loc = hover;
void update_setpoint(double x, double y) {
    pos = *; //-- assign non-deterministic value
    if (loc == hover) {
        assume(INV_hover); assert(REQ_hover_wait);
        spnxt = (x,y); assert(INV_wait);
        loc = wait; return;
    }
    assert(0);
}
```
Discharging Premise 1

Verification Stubs for API functions

Application Source code that calls API functions

CBMC Software Model Checker

Verification Result

SUCCESS

```c
void A1() {
    for (int n=1;++n) {
        update_setpoint(n,0);
        while(!has_arrived());
    }
}
```
Discharging Premise 1

Verification Stubs for API functions

Application Source code that calls API functions

CBMC Software Model Checker

Verification Result

FAILURE

More details in paper

```c
void A2() {
    for (int n=1;; ++n) {
        update_setpoint(n, 0);
        while(has_arrived());
    }
}
```
Discharging Premise 2

Hybrid Automaton extracted from contract automaton

Hybrid automaton for controller dynamic

SpaceEX Hybrid Reachability Tool

Verification Result
Discharging Premise 2

SpaceEX Hybrid Reachability Tool

Success

Used symmetry to reduce statespace (dimensions, time horizon)

More details in paper
Verifying Distributed Collision Avoidance

We implemented a system with 10 quadcopters moving on the 2D grid using a DSL called DMPL that supports synchronous model of computation.

Verified two properties of this distributed system using software model checking:

- **Property 1.** Distinct quadcopters have disjoint `cellcur` and `cellnext` values
  - $\forall i \neq j \in [0,9]. \text{cellcur}[i] \neq \text{cellcur}[j] \land \text{cellcur}[i] \neq \text{cellnext}[j]$
- **Property 2.** Setpoints are 5 times cell values
  - $spcur = 5 \times cellcur$ and $spnxt = 5 \times cellnext$

17.5KLOC, 2.9GHz, 16GB RAM, 1900 seconds

Proved that these two properties and the property of movement of a single quadcopter verified earlier using a contract automaton $\Rightarrow$ distance between centers of distinct quadcopters is always greater than the quadcopter diameter:

- Encoded as a SMT formula and proved using Z3
- Implies physical collision avoidance of the distributed system
Conclusion

Presented a compositional approach to verify CPS consisting of an application and a controller

• Combine software model checking with hybrid system reachability and works at the source code level
• Based on a contract automaton to capture application-controller dependencies and a sounds assume-guarantee style proof rule
• Validated on a multi-agent collision avoidance protocol

Future Work

• Manual steps automated and packaged as an end-to-end tool
• Parametric verification can reason about unbounded number of quadcopters and grids
QUESTIONS?