The ComFoRT Reasoning Framework

Sagar Chaki
James Ivers
Natasha Sharygina
Kurt Wallnau

© 2005 by Carnegie Mellon University
Predictable Assembly from Certifiable Components

Enable the development of software systems from software components where:

- critical runtime attributes e.g., performance and safety, are reliably predicted (predictable assembly)

- properties of software components needed for prediction are trusted (certifiable components)
The Construction and Composition Language (CCL) formalizes this idiom.
PACC Reasoning Frameworks

- development of analysis techniques
- transitioning of analysis to practitioners
ComFoRT Reasoning Framework

• Contains a software model checker Copper:
  - provides new model checking techniques developed for verification of component software
  - builds on academic tool MAGIC

• Analysis models are automatically extracted from programs

• Claims and verification results (counterexamples) are mapped to programs
Verification Domain

High-level designs (CCL programs) and C programs

• Sequential and concurrent

Communication via shared actions

• Synchronous communication

• Asynchronous execution
Copper Capabilities

State/Event-based Verification

• leverages distinction between *data* and *communication actions*

Compositional Deadlock Detection

• automated deadlock detection that ensures *sound abstractions* and acts as a space reduction procedure

Verification of Evolving Systems

• automated component *substitutability checks*
ComFoRT Underlying Framework

- CCL/C Program
- Abstraction
- Model Checker
- Validation
- Refinement

- No error or bug found
- Counterexample
- Validation successful
- Bug found
- Spurious counterexample

© 2005 by Carnegie Mellon University
ComFoRT Underlying Framework

- **CCL/C Program**
- **Abstraction**
- **Refinement**
- **Model Checker**
- **Validation**

- **No error or bug found**
- **Counterexample**
- **Validation successful**
- **Bug found**
- **Spurious counterexample**
State/Event-based Model Checking (IFM04)

Labeled Kripke Structures

- Every state is labeled with a set of atomic propositions, \( P \), true in the state
- Every LKS comes with an alphabet of actions, \( \Sigma \)

State/Event LTL and State/Event AW formalisms

Efficient model checking algorithms for SE-LTL and SE-AW employing the compositional abstraction-refinement framework

© 2005 by Carnegie Mellon University
State/Event-based Model Checking

Labeled Kripke Structures

- Every state is labeled with a set of atomic propositions, $P$, true in the state
- Every LKS comes with an alphabet of actions, $\Sigma$

State/Event LTL and State/Event AW formalisms

Surge Protector: State/Event

Changes of current beyond threshold are disallowed

\[ G ((c2 \rightarrow m=2) \& (c1 \rightarrow (m=1 \lor m=2))) \]
Surge Protector : State Only

\[ G (((c=0 \lor c=2) \land X (c=1)) \rightarrow (m=1 \lor m=2)) \land \]

\[ G (((c=0 \lor c=1) \land X (c=2)) \rightarrow m=2) \]
Deadlocks are not preserved by abstraction

- Abstraction refinement does not work

\[
\text{Copper: } \quad \text{Deadlock} = \text{AbsRef}(s) = \Sigma
\]

To preserve deadlock, the abstract model over-approximates not just what concrete program can do but also what it refuses.
Compositional Deadlock Detection

Deadlock is inherently non-compositional

• Can’t say anything by looking at components individually

\textit{Copper: } \text{AbsRef}(A_1, A_2) = \text{AbsRef}(A_1) \cup \text{AbsRef}(A_2)

Abstract deadlock - reachable state \( s \) such that \( \text{AbsRef}(s) = \Sigma \)

\textit{Copper: No abstract deadlock in abstract models \quad No deadlock in concrete models}

\textbf{Automated, compositional and iterative deadlock detection.} In Proceedings of the \textit{Conference on Formal Methods for Codesign (MEMOCODE) 2004}, by Sagar Chaki, Edmund Clarke, Joel Ouaknine and Natasha Sharygina
Component Substitutability Check

Containment check (Local correctness)

Are all local old services (properties) of the verified component contained in the upgraded component?
Component Substitutability Check

Compatibility Check (Global safety check)

Are new services of the upgraded component safe with respect to other components in assembly: all global specifications still hold?
Substitutability Check Approach

• Procedure for checking simultaneous upgrades of multiple components (FM’04)
  - Abstraction (under- and over- approximations) for the component containment check
  - Compositional reasoning + learning regular sets for automated compatibility check

• Procedure for checking individual component upgrades (SAVCBS’04)
  - Algorithms based on learning regular sets technique for the component containment and compatibility tests
Substitutability Check Approach

• Procedure for checking *simultaneous upgrades* of *multiple components*

  - Abstraction (under- and over- approximations) for the component containment check

  - Compositional reasoning + learning regular sets for automated compatibility check


**Verification of Evolving Software**, In Proceedings of SAVCBS 2004 by Sagar Chaki, Natasha Sharygina and Nishant Sinha
int numWaiting = 0;
int waiting1 = 0;
int type1 = 0;
int waiting2 = 0;
int type2 = 0;
int random = 0;
int error = 0;
int caller = 0;

sink mutex EnterCriticalSection_read (consume int caller);
sink mutex LeaveCriticalSection_read (consume int caller);
sink mutex EnterCriticalSection_write (consume int caller);
sink mutex LeaveCriticalSection_write (consume int caller);

threaded react C3 (EnterCriticalSection_read, LeaveCriticalSection_read, EnterCriticalSection_write, LeaveCriticalSection_write);
start -> one ()
one -> two (trigger ^EnterCriticalSection_read(caller));
two -> one ()
one -> three (trigger ^LeaveCriticalSection_read(caller));
three -> one ()
one -> four (trigger ^EnterCriticalSection_write(caller));
four -> one ()
one -> five (trigger ^LeaveCriticalSection_write(caller));
five -> one ()

state two / entry/
int numWaiting = 0;
int waiting1 = 0;
int type1 = 0;
int waiting2 = 0;
int type2 = 0;
int random = 0;
int error = 0;
int caller = 0;

sink mutex EnterCriticalSection_read (consume int caller);
sink mutex LeaveCriticalSection_read (consume int caller);
sink mutex EnterCriticalSection_write (consume int caller);
sink mutex LeaveCriticalSection_write (consume int caller);

threaded react C3 (EnterCriticalSection_read, LeaveCriticalSection_read, EnterCriticalSection_write, LeaveCriticalSection_write)
start -> one ()
one -> two (trigger ^EnterCriticalSection_read(caller));
two -> one ()
one -> three (trigger ^LeaveCriticalSection_read(caller));
three -> one ()
one -> four (trigger ^EnterCriticalSection_write(caller));
four -> one ()
one -> five (trigger ^LeaveCriticalSection_write(caller));
five -> one ()

state two / entry;

Verification complete.

Claim Claim28 does not hold. See targets_CL Claim28.txt for the counterexample.
abstract abs?, (PO::write_msg_queue.queue == 0) \&\& (PO::write_msg_queue_msg == 0),
  (P1::read_msg_queue.queue == 0) \&\& (P1::read_msg_queue_msg == 0) \&\& (P1::ipc_queue_writeTimeout == 0) \&\& (P2::ipc_queue_readTimeout == 0) \&\& (P2::ipc_queue_num == 0) \&\& (P2::ipc_queue_size == 2) \&\& (P2::ipc_queue_message1 == 0) \&\& (P2::ipc_queue_source1 == 0) \&\& (P2::ipc_queue_message2 == 0) \&\& (P2::ipc_queue_source2 == 0) \&\& (P2::ipc_queue_error == 0) \&\& (P3::critical_section_owner == 0) \&\& (P3::critical_section_ownerType == 0) \&\& (P3::critical_section_waiting1 == 0) \&\& (P3::critical_section_waiting2 == 0) \&\& (P3::critical_section_type2 == 0) \&\& (P3::critical_section_random == 0) \&\& (P3::critical_section_error == 0) \&\& (P3::critical_section_size == 0).

//Claim0: G[[P3::critical_section_error == 0]]; Claim0 = (epsilon \rightarrow ERROR0).
Claim0 = (P3::critical_section_error == 0).

//Claim1: G[[begin_WaitForSingleObject_write \rightarrow [P2::ipc_queue_numMessages == P2::ipc_queue_size]]]; Claim1 = (epsilon \rightarrow S11).
Claim1 = (begin_WaitForSingleObject_write \rightarrow ERROR1).
Claim1 = (P2::ipc_queue_numMessages == P2::ipc_queue_size).

Claim2: G[[P2::ipc_queue_error == 0]]; Claim2 = (epsilon \rightarrow ERROR2).
Claim2 = (P2::ipc_queue_error == 0).

Claim3: G[[P3::critical_section_numWaiting < 2]]; Claim3 = (epsilon \rightarrow ERROR3).
Claim3 = (P3::critical_section_numWaiting < 2).

4 well-formed claims found.
### Pre-processing complete.

Starting verification as background task...
model extracted in 27638.8 milliseconds ...
model loaded from file ...
implementation states for control locations computed ...
model extracted in 443.5 milliseconds ...
model saved in 73.6 milliseconds ...
implementation states for control locations computed ...
model extracted in 27862.0 milliseconds ...

<<< CHECKPOINT : new predicates inferred from spurious counterexample >>>
Seeding with branch ( P3::critical_section_owner == 0 )
Seeding with branch ( P3::critical_section_owner == P3::critical_section_caller )
Seeding with branch ( P3::critical_section_owner == 0 )
Seeding with branch ( P3::critical_section_owner == P3::critical_section_caller )

<<< END CHECKPOINT >>>

spurious counter-example detected !!
abstraction refined !!
starting iteration number 3 ...
model saved in 75.7 milliseconds ...
number of abstract implementation states = 3975
implementation states for control locations computed ...
action-guided transitions computed ...
model extracted in 25843.6 milliseconds ...
implementation machine extracted in 25844.2 milliseconds ...
global states : | 260 264 526 3975 7 7 2 | = 1989161201600
<<< CHECKPOINT : valid CF found >>>
CE dag projections analysed ... convernance relation does not exist !! abstraction absl is invalid ...
<<< END CHECKPOINT >>>

<<< CHECKPOINT : various statistics >>>
total global time = 1534534.2 milliseconds
total cpu time = 1414160.0 milliseconds
total input processing time = 542.7 milliseconds
total Euchi automaton construction time = 0.0 milliseconds
total implementation machine extraction time = 223985.4 milliseconds
total verification time = 1230970.9 milliseconds
total proof generation time = 0.0 milliseconds
total abstraction refinement time = 121052.3 milliseconds
total CE generation time = 12.6 milliseconds
total CE verification time = 2567.9 milliseconds
total predicate abstraction refinement time = 118444.3 milliseconds
total LTS abstraction refinement time = 0.0 milliseconds
total number of eliminating combinations = 0
max number of eliminating combinations for a CE = 0
max size of eliminating combination = 0
max size of tried combination = 0
number of iterations = 4
number of predicate iterations = 4
number of lts iterations = 0
number of seed branches : 5
specification details : 3 states 2 transitions
number of implementation states : 1064216282400
Applications

IPC Module
• Deployed by a world leader in robotics
• Discovered synchronization bug under which senders would receive the wrong answer to their requests
• Problem had remained undetected for seven years prior to independent discovery by business unit

Case Study: Micro-C OS
• Real-time OS for embedded applications
  - 6000+ LOC, widely used
• Verified locking discipline
• Found four bugs
  - Missing unlock and return
  - Three already reported
Ongoing and Future Work

• Use a SAT solver for computing abstraction
  - Semantics of bit-wise operators is taken into account

• Use of pattern languages for specifying properties

• Integrated Abstraction and Compositional reasoning

• Component certification
ComFoRT Resources

ComFoRT tools
• http://www.sei.cmu.edu/pacc/comfort.html

Ongoing industrial & academic collaborations

• Prof. Edmund Clarke and his model checking group, Prof. Peter Lee at CMU
• Prof. Dr. Daniel Kroening from ETH Zurich
• Industrial corporate research centers developing embedded controllers

Conference and Journal publications
References


References

**Verification of Evolving Software**, In Proceedings of SAVCBS 2004 by Sagar Chaki, Natasha Sharygina and Nishant Sinha

**Snapshot of CCL: A Language for Predictable Assembly**, In CMU/SEI TR-2002-TR-031, by James Ivers and Kurt Wallnau

**A Technology for Predictable Assembly from Certifiable Components (PACC)**, In CMU/SEI-TR-2003-TR-009, by Kurt Wallnau