Verification of Parameterized Concurrent Programs
By Modular Reasoning about Data and Control

Zachary Kincaid    Azadeh Farzan

University of Toronto

January 18, 2013
Parameterized concurrent programs

Goal

Compute numerical invariants (e.g. intervals, octagons, polyhedra) for parameterized concurrent programs.

Solution: annotation $\nu$ such that if some thread $T$’s program counter is at $v$, then $\nu(v)$ holds over the globals & locals of $T$. 
Parameterized concurrent programs

Goal

Compute numerical invariants (e.g. intervals, octagons, polyhedra) for parameterized concurrent programs.
Solution: annotation $\iota$ such that if some thread $T$’s program counter is at $v$, then $\iota(v)$ holds over the globals & locals of $T$.

Our program model has:

- **Unbounded concurrency**: program is the parallel composition of $n$ copies of some thread $T$, where $n$ is a parameter
  - Invariants must be sound for all $n$
Parameterized concurrent programs

**Goal**

Compute numerical invariants (e.g. intervals, octagons, polyhedra) for parameterized concurrent programs.

Solution: annotation $\iota$ such that if some thread $T$’s program counter is at $v$, then $\iota(v)$ holds over the globals & locals of $T$.

Our program model has:

- **Unbounded concurrency**: program is the parallel composition of $n$ copies of some thread $T$, where $n$ is a parameter
  - Invariants must be sound for all $n$
- **Unbounded data domains**
Parameterized concurrent programs

Goal

Compute numerical invariants (e.g. intervals, octagons, polyhedra) for parameterized concurrent programs.

Solution: annotation \( \iota \) such that if some thread \( T \)'s program counter is at \( v \), then \( \iota(v) \) holds over the globals & locals of \( T \).

Our program model has:

- **Unbounded concurrency**: program is the parallel composition of \( n \) copies of some thread \( T \), where \( n \) is a parameter
  - Invariants must be sound for all \( n \)
- **Unbounded data domains**

Natural model for device drivers, file systems, client/server-type programs, ...
We develop an attack on the parameterized verification problem based on separating it into a data module and a control module.

- Data module computes numerical invariants
- Control module computes a program model
Contributions

1. We develop an attack on the parameterized verification problem based on separating it into a **data** module and a **control** module
   - **Data module** computes numerical invariants
   - **Control module** computes a program model

2. We propose *data flow graphs* as a program representation for (parameterized) concurrent programs
Contributions

1. We develop an attack on the parameterized verification problem based on separating it into a data module and a control module
   - **Data module** computes numerical invariants
   - **Control module** computes a program model

2. We propose *data flow graphs* as a program representation for (parameterized) concurrent programs

3. We give a semicompositional algorithm for constructing data flow graphs
Contributions

1. We develop an attack on the parameterized verification problem based on separating it into a data module and a control module
   - **Data module** computes numerical invariants
   - Control module computes a program model

2. We propose *data flow graphs* as a program representation for (parameterized) concurrent programs

3. We give a semicompositional algorithm for constructing data flow graphs
Sequential program analysis

- Flow analysis: solve a system of equations valued over some abstract domain
- For sequential programs, equations come from the control flow graph:

\[
egin{align*}
IN(t) &= \top \\
OUT(t) &= [t](IN(t)) \\
IN(u) &= OUT(t) \lor OUT(w) \\
OUT(u) &= [u](IN(u)) \\
IN(v) &= OUT(t) \\
OUT(v) &= [v](IN(v)) \\
IN(w) &= OUT(u) \lor OUT(v) \\
OUT(w) &= [w](IN(w))
\end{align*}
\]
Sequential program analysis

- Flow analysis: solve a system of equations valued over some abstract domain
- For sequential programs, equations come from the control flow graph:

\[
IN(t) = \top \\
OUT(t) = [t](IN(t))
\]

\[
IN(u) = OUT(t) \lor OUT(w) \\
OUT(u) = [u](IN(u))
\]

\[
IN(v) = OUT(t) \\
OUT(v) = [v](IN(v))
\]

\[
IN(w) = OUT(u) \lor OUT(v) \\
OUT(w) = [w](IN(w))
\]

- How about parameterized programs?
Data flow

Represent **data flow**, not control flow:
Represent **data flow**, not control flow:
Data flow

Represent **data flow**, not control flow:
Why data flow?

Invariant: $x = 0$

$y := 0$

acquire(lock)

assert($x = 0$)

release(lock)

acquire(lock)

$x := 1$

$x := 0$

release(lock)

Break invariant

Restore invariant
A DFG for a program $P$ is a directed graph $P^\# = \langle Loc, \rightarrow \rangle$, where

- $\rightarrow \subseteq Loc \times Vars \times Loc$ is a set of directed edges labeled by program variables
- $Loc$ contains a distinguished uninit vertex
- Note: # of vertices does not depend on # of threads
Representing traces

- A program is *represented* by a DFG $P^\#$ if all its feasible traces are represented by $P^\#$. 

- A trace is *represented* by a DFG $P^\#$ if all data flow edges it witnesses belong to $P^\#$. 

- A trace witnesses a data flow $u \rightarrow x \rightarrow v$ iff it is of the form: 

  $\left( x_{\text{local}} \Rightarrow \text{requires} n = m \right)$
Representing traces

• A program is *represented* by a DFG $P^\#$ if all its feasible traces are represented by $P^\#$.

• A trace is *represented* by a DFG $P^\#$ if all data flow edges it witnesses belong to $P^\#$. 

Representing traces

- A program is *represented* by a DFG $P^\sharp$ if all its feasible traces are represented by $P^\sharp$.
- A trace is *represented* by a DFG $P^\sharp$ if all data flow edges it witnesses belong to $P^\sharp$.
- A trace *witnesses* a data flow $u \rightarrow^x v$ iff it is of the form:

$$\langle T_n, u \rangle$$

Thread $n$ executes $u$, $u$ modifies $x$

Thread $m$ at $v$

No modifications to $x$

($x$ local $\Rightarrow$ requires $n = m$)
Representing traces

- A program is *represented* by a DFG $P^\#$ if all its feasible traces are represented by $P^\#$.
- A trace is *represented* by a DFG $P^\#$ if all data flow edges it witnesses belong to $P^\#$.
- A trace *witnesses* a data flow $u \rightarrow^x v$ iff it is of the form:

```
x := x + 1
y := 1
x := x + y
x := -x
```

Thread $n$ executes $u$, $u$ modifies $x$

$(x \text{ local } \Rightarrow \text{ requires } n = m)$

Thread $m$ at $v$

No modifications to $x$
Computing invariants with DFGs

- DFGs induce a set of equations:

\[
\begin{align*}
IN(v)_x &= \bigvee_{u \rightarrow x v} \exists (V ars \setminus \{x\}).OUT(u) \\
IN(v) &= \bigwedge_{x \in Var} IN(v)_x \\
OUT(v) &= \llbracket v \rrbracket(IN(v))
\end{align*}
\]

- Define an *inductive annotation* to be a solution to these equations.
Computing invariants with DFGs

- DFGs induce a set of equations:

\[
\begin{align*}
IN(v)_x &= \bigvee_{u \rightarrow x v} \exists (Vars \setminus \{x\}).OUT(u) \\
IN(v) &= \bigwedge_{x \in Var} IN(v)_x \\
OUT(v) &= \llbracket v \rrbracket(IN(v))
\end{align*}
\]

- Define an *inductive annotation* to be a solution to these equations.

**Theorem (DFG soundness)**

*If* $\sigma$ *is a trace represented by a DFG* $P^\#$, *and* $\iota$ *is an inductive annotation for* $P^\#$, *then* $\iota$ *safely approximates the states reached by* $\sigma$.
Overview

Data module

Control module
Constructing data flow graphs

Goal

Compute the set of all \( \langle u, x, v \rangle \) such that there is some feasible trace that witnesses \( u \rightarrow^x v \)

- Strategy:
  - Overapproximate the set of feasible traces
  - Compute dataflow edges witnessed by one of these traces
Precise DFG construction needs data

(flag is initially 0)

```
assume(flag)
assert(x != null)
x := null
x := alloc(...) 
flag := 1
```
Precise DFG construction needs data

(flag is initially 0)

\[
\begin{align*}
\text{assume}(\text{flag}) \\
\text{assert}(x \neq \text{null}) \\
\text{x := null} \\
\text{x := alloc(...)} \\
\text{flag := 1} \\
\end{align*}
\]
Precise DFG construction needs data

(flag is initially 0)

- `assume(flag)`
- `assert(x != null)`
- `x := null`
- `x := alloc(...)`
- `flag := 1`

 cannot execute!
\( \nu \)-feasible traces

Use an annotation \( \nu \) to rule out infeasible traces: a trace \( \sigma \) is \( \nu \)-infeasible if there is some subtrace \( \sigma' \langle T_n, v \rangle \), some thread \( m \), and some location \( u \) such that

- Thread \( m \) is at location \( u \) after executing \( \sigma' \)
- Thread \( n \) may not execute \( v \) in any state satisfying \( \nu(u) \).
\(\nu\)-feasible traces: example

\(\text{flag is initially 0)}\)

\begin{align*}
\text{assume(} \text{flag})
\Rightarrow & \text{assert(} x \neq \text{null})
\Rightarrow & x := \text{null}
\Rightarrow & x := \text{alloc(...)}
\Rightarrow & \text{flag := 1}
\end{align*}
\(\nu\)-feasible traces: example

(flag is initially 0)

\[\begin{align*}
\text{assume}(\text{flag}) & \quad \text{assert}(x \neq \text{null}) \\
& \quad \text{x} := \text{null} \\
& \quad \text{x} := \text{alloc}(\ldots) \\
& \quad \text{flag} := 1
\end{align*}\]

\[\langle T_1, x := \text{null} \rangle \quad \langle T_2, \text{assume}(\text{flag}) \rangle \quad \langle T_2, \text{assert}(x \neq \text{null}) \rangle\]
\( \nu\)-feasible traces: example

(flag is initially 0)

```
assume(flag)

assert(x != null)
```

\[ x := \text{null} \]

\[ x := \text{alloc(...)} \]

\[ \text{flag := 1} \]

```
\langle T_1, x := \text{null} \rangle
```

\[ \langle T_2, \text{assume(flag)} \rangle \]

\[ \langle T_2, \text{assert(x != null)} \rangle \]

\( \text{guard: flag \neq 0} \)

\( T_1 \text{ at } x := \text{alloc(...)} \)

\( T_2 \text{ at } \text{assume(flag)} \)

\( \text{is } \nu(x := \text{alloc(...)}) \land \text{flag \neq 0} \)

\( \text{satisfiable?} \)
\(\nu\)-feasible traces: example

(flat is initially 0)

\begin{align*}
&\text{assert}(x \neq \text{null}) \\
&x := \text{null} \\
x := \text{alloc}(\ldots) \\
&\text{guard}: \text{flag} \neq 0 \\
&T_1 \text{ at } x := \text{alloc}(\ldots) \\
&T_2 \text{ at } \text{assume}(\text{flag}) \\
&\text{is } \nu(x := \text{alloc}(\ldots)) \land \text{flag} \neq 0 \\
&\text{satisfiable?}
\end{align*}

- \(\nu(x := \text{alloc}(\ldots)): \text{flag} = 0 \Rightarrow \text{infeasible}\)
\( \nu \)-feasible traces: example

\[\begin{align*}
\text{assume}(\text{flag}) \\
\text{assert}(x \neq \text{null}) \\
x := \text{null} \\
x := \text{alloc}(\ldots) \\
\text{flag} := 1
\end{align*}\]

\[\begin{align*}
\langle T_1, x := \text{null} \rangle \\
\langle T_2, \text{assume}(\text{flag}) \rangle \\
\langle T_2, \text{assert}(x \neq \text{null}) \rangle
\end{align*}\]

\text{guard: } \text{flag} \neq 0
\text{\(T_1\) at } x := \text{alloc}(\ldots)
\text{\(T_2\) at } \text{assume}(\text{flag})
is \nu(x := \text{alloc}(\ldots) \land \text{flag} \neq 0) \text{ satisfiable?}

- \(\nu(x := \text{alloc}(\ldots)): \text{flag} = 0 \Rightarrow \text{infeasible}\)
- \(\nu(x := \text{alloc}(\ldots)): \text{true} \Rightarrow \text{feasible}\)
Constructing data flow graphs

**Goal**

Compute the set of all $\langle u, x, v \rangle$ such that there is some feasible trace that witnesses $u \rightarrow^x v$

- **Strategy:**
  - Overapproximate the set of feasible traces
  - Compute dataflow edges witnessed by one of these traces
Constructing data flow graphs

Goal

Compute the set of all \( \langle u, x, v \rangle \) such that there is some feasible trace that witnesses \( u \rightarrow^x v \)

• Strategy:
  ✓ Overapproximate the set of feasible traces by \( \iota \)-feasible traces
  • Compute dataflow edges witnessed by one of these traces
Constructing data flow graphs

**Goal**

Compute the set of all $\langle u, x, v \rangle$ such that there is some feasible trace that witnesses $u \rightarrow^x v$

- **Strategy:**
  - ✓ Overapproximate the set of feasible traces by $\Upsilon$-feasible traces
  - Compute dataflow edges witnessed by one of these traces
    - Parameterization is still an obstacle
Constructing data flow graphs

Goal

Compute the set of all $\langle u, x, v \rangle$ such that there is some feasible trace that witnesses $u \rightarrow^x v$

- Strategy:
  - ✓ Overapproximate the set of feasible traces by $\iota$-feasible traces
  - Compute dataflow edges witnessed by one of these traces
    - Parameterization is still an obstacle
    - Data flow edges for 2-thread $\iota$-feasible witnesses can be computed efficiently
Lemma (projection)

Let $\iota$ be an annotation, let $\sigma$ be an $\iota$-feasible trace, and let $N$ be a set of threads. Then $\sigma\upharpoonright N$, the projection of $\sigma$ onto $N$, is also $\iota$-feasible.

Thread $n$ executes $u$, $u$ modifies $x$

Thread $m$ at $v$

No modifications to $x$
Lemma (projection)

Let $\iota$ be an annotation, let $\sigma$ be an $\iota$-feasible trace, and let $N$ be a set of threads. Then $\sigma|_N$, the projection of $\sigma$ onto $N$, is also $\iota$-feasible.
Lemma (projection)

Let $\iota$ be an annotation, let $\sigma$ be an $\iota$-feasible trace, and let $N$ be a set of threads. Then $\sigma|_N$, the projection of $\sigma$ onto $N$, is also $\iota$-feasible.

- A data flow edge $u \xrightarrow{x} v$ has an $\iota$-feasible witness iff it has a 2-thread $\iota$-feasible witness.
Feedback loop

- Given a DFG, we know how to compute numerical invariants
- Given numerical invariants, we know how to compute a DFG
Feedback loop

- Given a DFG, we know how to compute numerical invariants
- Given numerical invariants, we know how to compute a DFG

Sequential reaching definitions

Sequential DFG

Data analysis

DFG construction

Annotation
Feedback loop

- Given a DFG, we know how to compute numerical invariants
- Given numerical invariants, we know how to compute a DFG

Sequential reaching definitions

Data flow edges

Sequential DFG

Data analysis

DFG construction

Annotation
Feedback loop

- Given a DFG, we know how to compute numerical invariants
- Given numerical invariants, we know how to compute a DFG
Given a DFG, we know how to compute numerical invariants
Given numerical invariants, we know how to compute a DFG
Feedback loop

- Given a DFG, we know how to compute numerical invariants
- Given numerical invariants, we know how to compute a DFG
Experimental results

- We implemented our algorithm in a tool, **DUET**
- Integer overflow & array bounds checks for 15 Linux device drivers
  - **DUET** proves 1312/1597 (82%) assertions correct in 13m9s
Experimental results: Boolean programs

Boolean abstractions of Linux device drivers:

<table>
<thead>
<tr>
<th>Suite 1</th>
<th>DUET</th>
<th>Linear interfaces</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertions proved</td>
<td>2503</td>
<td>1382</td>
<td>81% increase</td>
</tr>
<tr>
<td>Average time</td>
<td>3.4s</td>
<td>16.9s</td>
<td>5x speedup</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suite 2</th>
<th>DUET</th>
<th>Dynamic cutoff detection</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertions proved</td>
<td>55</td>
<td>19</td>
<td>189% increase</td>
</tr>
<tr>
<td>Average time</td>
<td>8.2s</td>
<td>24.9s</td>
<td>3x speedup</td>
</tr>
</tbody>
</table>

• Separate reasoning into a data module and a control module
Conclusion

- Separate reasoning into a data module and a control module
- Data flow graphs represent parameterized programs
Conclusion

- Separate reasoning into a data module and a control module
- Data flow graphs represent parameterized programs
- Semi-compositional DFG construction algorithm
Questions?

Thank you for your attention.
• Improved algorithms for inferring groups of related variables to improve DFGs analyses over relational domains (e.g., octagons, polyhedra)
• Extension to handle aliasing