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

Zachary Kincaid Azadeh Farzan

University of Toronto

January 30, 2012

Parameterized concurrent programs

Goal

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

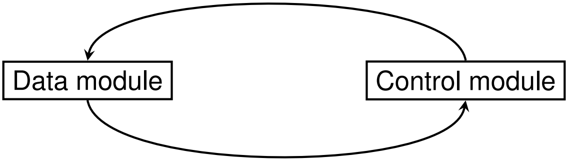
Our program model has:

- **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, ...

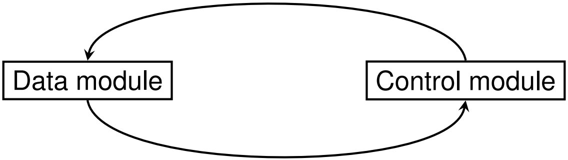
Contributions

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



Contributions

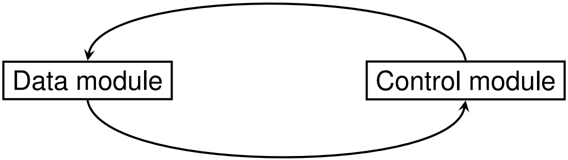
1. We develop an attack on the parameterized verification problem based on separating it into a **data** module and a **control** module
   - **Data module** computes numerical invariants
   - **Control module** computes a program model



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

Contributions

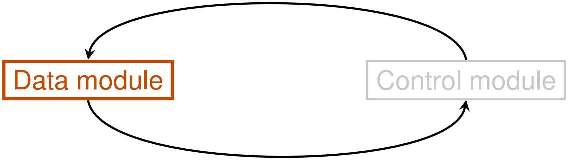
1. We develop an attack on the parameterized verification problem based on separating it into a **data** module and a **control** module
   - **Data module** computes numerical invariants
   - **Control module** computes a program model



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



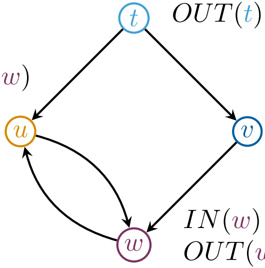
2. We propose *data flow graphs* as a program representation for (parameterized) concurrent programs
3. We give a semicompositional algorithm for constructing data flow graphs

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:

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

$$IN(u) = OUT(t) \vee OUT(w)$$
$$OUT(u) = [\![u]\!](IN(u))$$



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

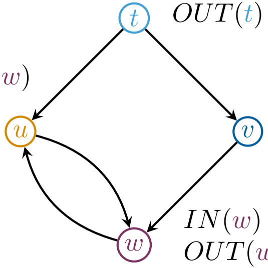
$$IN(w) = OUT(u) \vee OUT(v)$$
$$OUT(w) = [\![w]\!](IN(w))$$

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:

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

$$IN(u) = OUT(t) \vee OUT(w)$$
$$OUT(u) = [\![u]\!](IN(u))$$



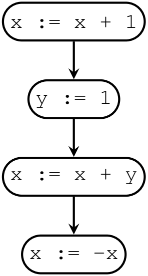
$$IN(v) = OUT(t)$$
$$OUT(v) = [\![v]\!](IN(v))$$

$$IN(w) = OUT(u) \vee OUT(v)$$
$$OUT(w) = [\![w]\!](IN(w))$$

- How about parameterized programs?

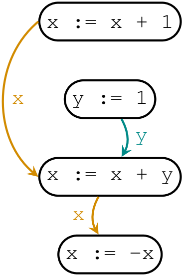
Data flow

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



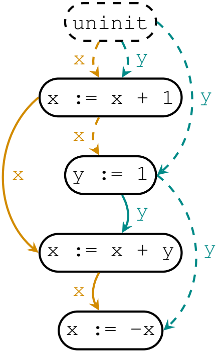
Data flow

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



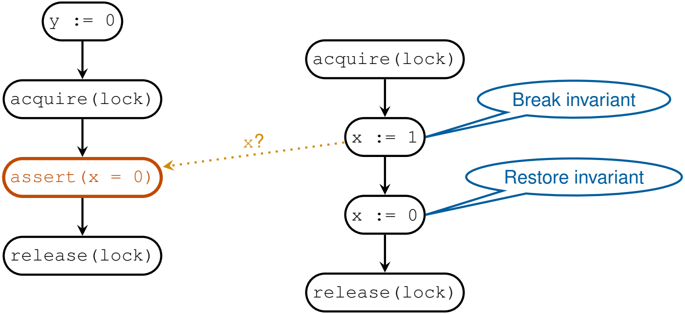
Data flow

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



Why data flow?

Invariant: $x = 0$



Data flow graphs

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

- $\rightarrow \subseteq Loc \times Vars \times Loc$ is a set of directed edges labeled by program variables

$$\boxed{\texttt{x := x + 1}} \xrightarrow{\texttt{x}} \boxed{\texttt{x := x + y}}$$

- Loc contains a distinguished `uninit` vertex
- Note: # of vertices does not depend on # of threads

Representing traces

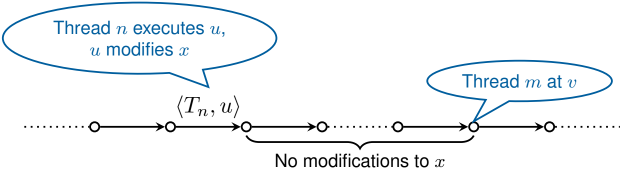
- A program is *represented* by a DFG $P^\sharp$ if all its feasible traces are represented by $P^\sharp$.

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 $P^\sharp$

Representing traces

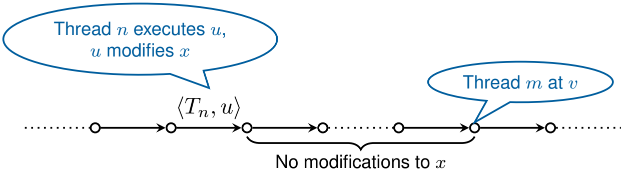
- A program is *represented* by a DFG $P^\sharp$ if all its feasible traces are represented by $P^\sharp$.
- A trace is *represented* by a DFG $P^\sharp$ if all data flow edges it witnesses belong to $P^\sharp$
- A trace *witnesses* a data flow $u \rightarrow^x v$ iff it is of the form:



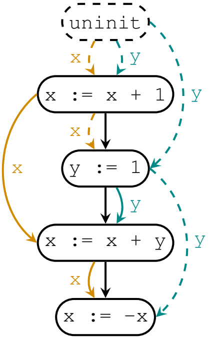
(x local $\Rightarrow$ requires $n = m$)

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:



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



Computing invariants with DFGs

- DFGs induce a set of equations: $$\boxed{\begin{aligned} IN(v)_x &= \bigvee_{u \to^x v} \exists (Vars \setminus \{x\}).OUT(u) \\ IN(v) &= \bigwedge_{x \in Var} IN(v)_x \\ OUT(v) &= [\![v]\!](IN(v)) \end{aligned}}$$
- Define an *inductive annotation* to be a solution to these equations.

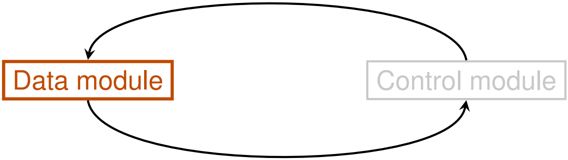
Computing invariants with DFGs

- DFGs induce a set of equations: $$\boxed{\begin{array}{l} IN(v)_x = \bigvee_{u \rightarrow^x v} \exists(Vars \setminus \{x\}).OUT(u) \\ IN(v) = \bigwedge_{x \in Var} IN(v)_x \\ OUT(v) = [\![v]\!](IN(v)) \end{array}}$$
- Define an *inductive annotation* to be a solution to these equations.

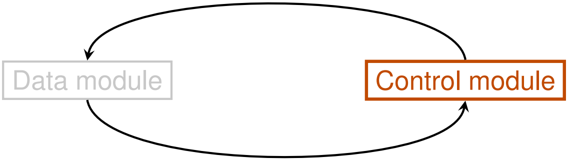
Theorem (DFG soundness)

If σ is a trace represented by a DFG $P^\sharp$, and ι is an inductive annotation for $P^\sharp$, then ι safely approximates the states reached by σ.

Overview



Overview



Constructing data flow graphs

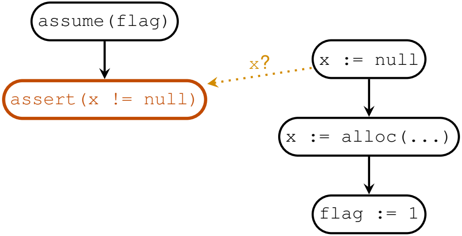
Goal

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

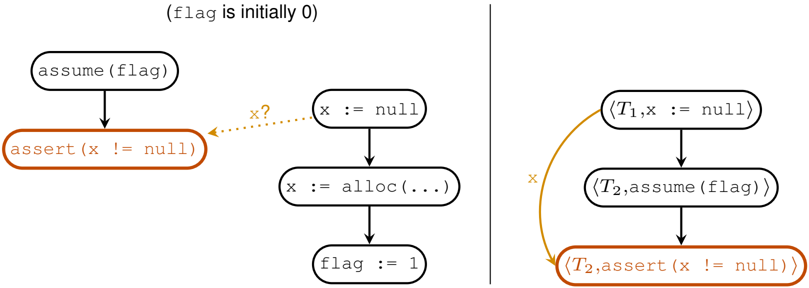
- Strategy:
  - Overapproximate the set of feasible traces
  - Compute dataflow edges witnessed by one of these traces

Precise DFG construction needs data

(`flag` is initially 0)

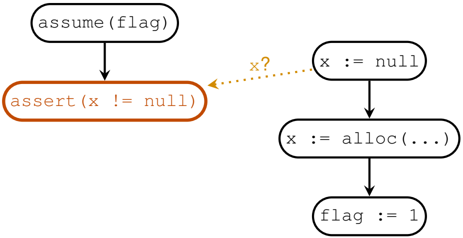


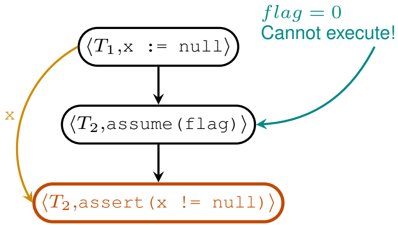
Precise DFG construction needs data



Precise DFG construction needs data

(`flag` is initially 0)





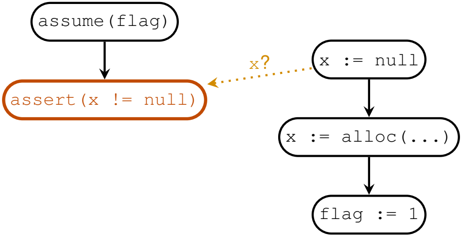
ι-feasible traces

Use an annotation ι to rule out infeasible traces: a trace σ is ι-*in*feasible 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 σ'
- Thread n may not execute v in any state satisfying $\iota(u)$.

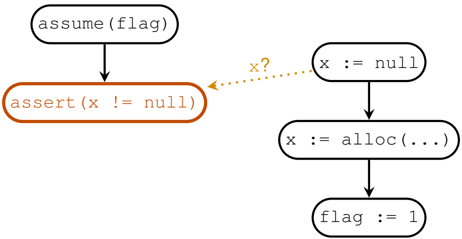
ι-feasible traces: example

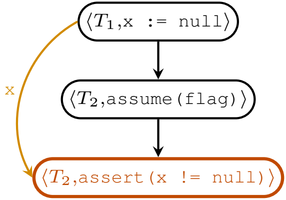
(`flag` is initially 0)



ι-feasible traces: example

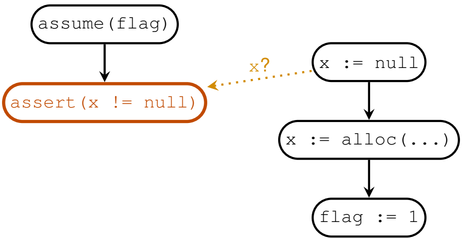
(`flag` is initially 0)

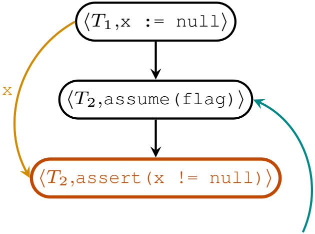




ι-feasible traces: example

(`flag` is initially 0)

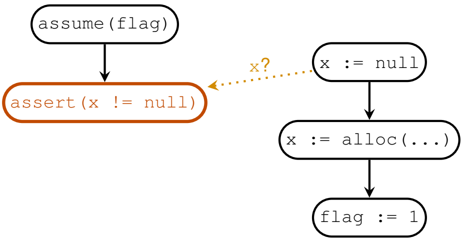


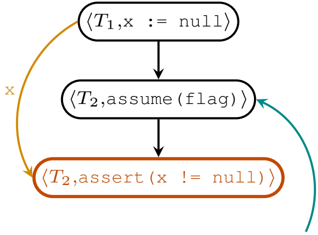


guard: $flag \neq 0$
T_1 at `x := alloc(...)`
T_2 at `assume(flag)`
is $\iota($`x := alloc(...)`$) \wedge flag \neq 0$
satisfiable?

ι-feasible traces: example

(`flag` is initially 0)



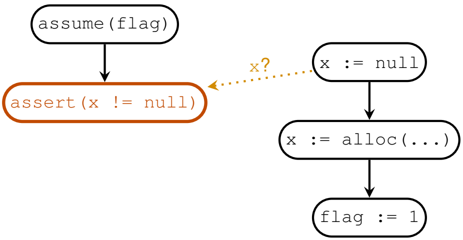


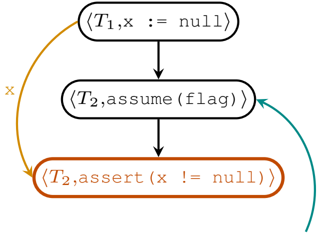
guard: $flag \neq 0$
T_1 at `x := alloc(...)`
T_2 at `assume(flag)`
is ι(`x := alloc(...)`) $\wedge\ flag \neq 0$
satisfiable?

- ι(`x := alloc(...)`) $: flag = 0 \Rightarrow$ infeasible

ι-feasible traces: example

(`flag` is initially 0)





guard: $flag \neq 0$
T_1 at `x := alloc(...)`
T_2 at `assume(flag)`
is $\iota($`x := alloc(...)`$) \wedge flag \neq 0$
satisfiable?

- $\iota($`x := alloc(...)`$) : flag = 0 \Rightarrow$ infeasible
- $\iota($`x := alloc(...)`$) : true \Rightarrow$ 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 $u \to^x v$

- 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 ι-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 ι-feasible traces
  - Compute dataflow edges witnessed by one of these traces
    - Parameterization is still an obstacle

Constructing data flow graphs

Goal

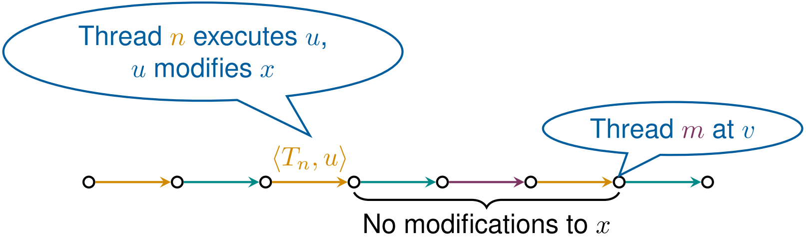
Compute the set of all $\langle u, x, v \rangle$ such that there is some feasible trace that witnesses $u \rightarrow^{x} v$

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

Projection

Lemma (projection)

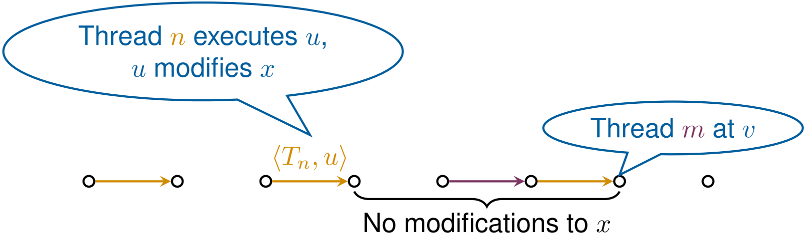
Let ι be an annotation, let σ be an ι-feasible trace, and let N be a set of threads. Then $\sigma|_N$, the projection of σ onto N, is also ι-feasible.



Projection

Lemma (projection)

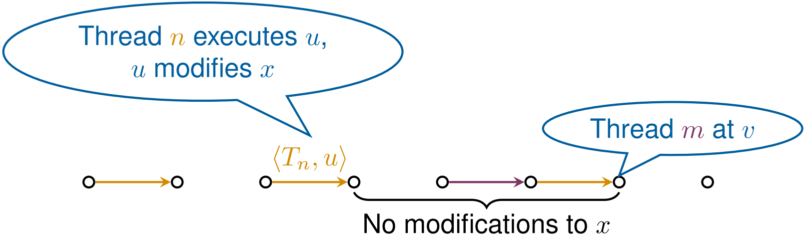
Let ι be an annotation, let σ be an ι-feasible trace, and let N be a set of threads. Then $\sigma|_N$, the projection of σ onto N, is also ι-feasible.



Projection

Lemma (projection)

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



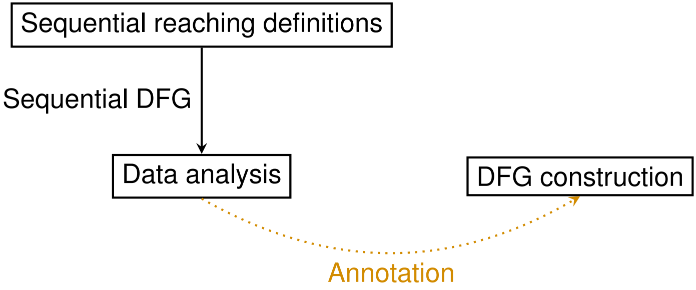
- A data flow edge $u \rightarrow^x v$ has an ι-feasible witness iff it has a 2-thread ι-feasible witness

Feedback loop

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

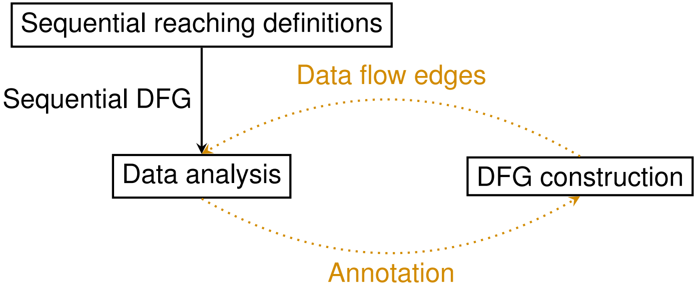
Feedback loop

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



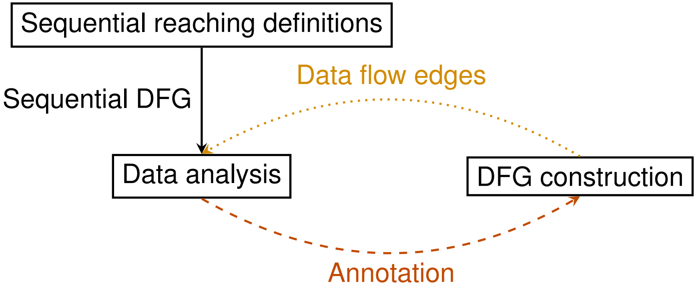
Feedback loop

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



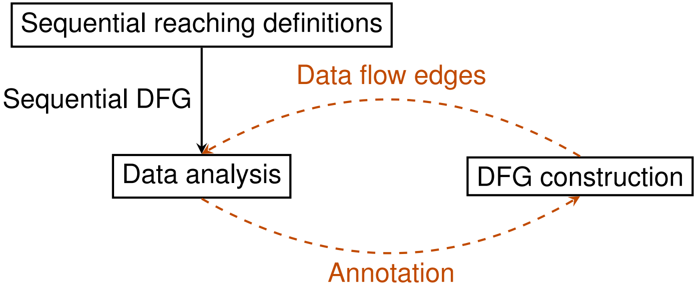
Feedback loop

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



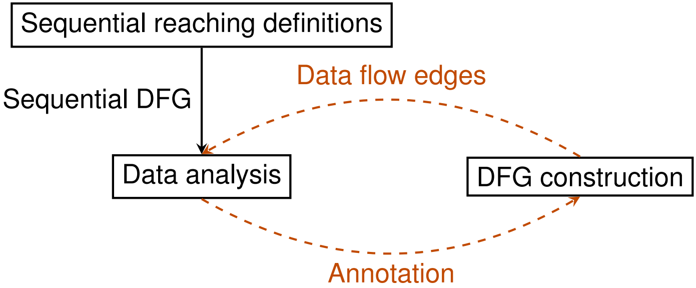
Feedback loop

- 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



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:

Suite 1	Duet	Linear interfaces[1]	Improvement
Assertions proved	2503	1382	81% increase
Average time	3.4s	16.9s	5x speedup

Suite 2	Duet	Dynamic cutoff detection[2]	Improvement
Assertions proved	55	19	189% increase
Average time	8.2s	24.9s	3x speedup

[1] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In CAV, pages 629–644. 2010.

[2] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In CAV, pages 645–659. 2010.

Conclusion

- 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.

Bonus slide: future work

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