Duet: Static Analysis for Unbounded Parallelism

Azadeh Farzan and Zachary Kincaid

University of Toronto

Abstract. Duet is a static analysis tool for concurrent programs in which the number of executing threads is not statically bounded. Duet has a modular architecture, which is based on separating the invariant synthesis problem in two subtasks: (1) data dependence analysis, which is used to construct a data flow model of the program, and (2) interpretation of the data flow model over a (possibly infinite) abstract domain, which generates invariants. This separation of concerns allows researchers working on data dependence analysis and abstract domains to combine their efforts toward solving the challenging problem of static analysis for unbounded concurrency. In this paper, we discuss the architecture of Duet as well as two data dependence analyses that have been implemented in the tool.

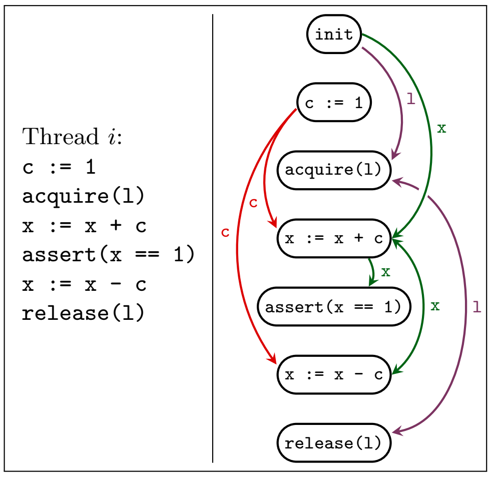
1 Introduction

Verification of concurrent programs is a notoriously challenging problem. The difficulty arises from the fact that the size of the control space of a concurrent program is exponential in the number of executing threads, which makes direct analysis of the control space infeasible. The problem is even more difficult for programs where the number of executing threads is not bounded: in this case, the control space is *infinite*.

In this paper, we present Duet, a static analysis tool for analyzing programs with *unbounded parallelism*. Duet is based on the general philosophy that *general invariant generation can be reduced to data-dependence analysis*. This philosophy is particularly interesting in the case of programs with unbounded parallelism, since in this setting invariant generation is a formidable problem but data-dependence analysis is tractable. In this paper, we expound on the philosophy of Duet. We describe its architecture and describe two different analyses implemented in the tool [4,5]. Finally, we present experimental results demonstrating the efficacy of our tool on a set device drivers.

2 Overview

Duet is based on the idea that much of the essential behaviour of a program is captured by the *data flow* of the program. Consider the program below, in which unboundedly many copies of the thread Thread i execute in parallel. A *data flow graph* (DFG) for this program is pictured to its right. The edges of this DFG match each read of a variable with the writes that *may* reach it: for example, the edge from `c := 1` to `x := x + c` labeled `c` indicates that the value of `c` may flow from `c := 1` to `x := x + c` (and thus, `c` *may* be `1` at `x := x + c`). Note that some edges in this graph go in the opposite direction of control flow (e.g., the double-headed edge between `x := x + c` and `x := x - c`, which indicates a pair of edges, one in each direction). The value of `x` may flow from `x := x - c` to `x := x + c` because these instructions may be executed by different threads. Notice also that several interesting edges *do not* exist: there is no `x`-labeled loop on `x := x - c`; Lock `l` enforces the atomicity of the lock block, which breaks this data flow; an occurrence of `x := x + c` must always be executed between any two `x := x - c` instructions.



Thread *i*:
```
c := 1
acquire(l)
x := x + c
assert(x == 1)
x := x - c
release(l)
```

A data flow graph represents a system of constraints that can be solved automatically (using standard techniques) to yield program invariants [5]. This is the essential idea of the Duet tool: since DFGs can be used to compute invariants, invariant generation can be reduced to constructing a DFG for a program. Moreover, since data flow captures an essential feature of program behaviour, this strategy is often sufficient to prove properties of interest. For the example program above, Duet is able to prove that assertion always holds by interpreting the DFG over an interval abstract domain.

3 Duet's Architecture

The high-level architecture of Duet is pictured in Figure 1. We will begin by describing the basic work-flow of Duet and then discuss each of the component modules in more detail.

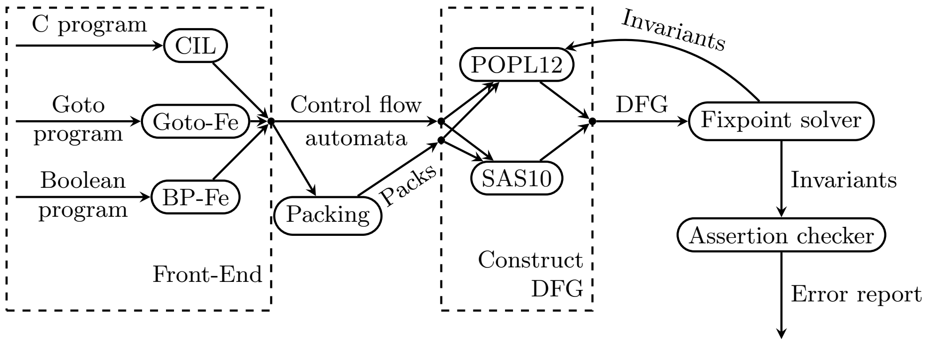


Fig. 1. Duet's architecture

First, one of the three **front-end** modules (CIL, Goto-Fe, BP-Fe) takes a (C, Goto, or Boolean) program as input and produces a system of control flow automata, with one automaton for each thread of the program. Next, a variable **packing** algorithm determines a set of *packs* (a *pack* is a set of semantically related variables, according to some heuristic). The set of packs and the system of control flow automata are fed into one of the two **construct DFG** modules (POPL12 or SAS10), which uses a data dependence analysis to construct a DFG for the program. The **fixpoint solver** interprets the DFG over some abstract domain to compute a set of invariants, which are passed to the assertion checker (in the case of POPL12, the invariants may alternatively be fed back into the DFG construction module). The **assertion checker** uses the invariants to determine check which assertions are safe and which *may* fail, and generates an error report.

3.1 Front-end

DUET accepts three types of inputs: (1) C programs using *pthreads* library for thread operations (using the CIL front-end [9]), (2) Boolean programs[1], or (3) goto programs, as produced by the GOTO-CC C/C++ front-end (part of the CPROVER project [1]). The front-end transforms these programs into a common form, namely a system of control flow automata. It also annotates the program with assertions, according to user input (for example, an assertion `assert(p != 0)` is generated for each access path of the form `*p`, if the `-check-null-pointer` option is set).

3.2 Variable Packing

In the DFGs implemented in DUET, each edge is labeled by a *pack* rather than a variable. A pack is a set of variables that may be related to one another. We may think of a DFG edge labeled by a pack as "carrying" a value for each variable in that pack (which allows an abstract domain to correlate variables belonging to a pack). The case where edges are labeled by variables is the special case where all packs are singleton sets. The **packing** module computes a set of packs from an input program, and passes those packs to the DFG construction phase. The choice of the abstract domain (non-relational vs relational) determines the choice of the packing algorithm used (there are currently two options in DUET).

3.3 DFG construction

A DFG construction module takes a system of control flow automata and a set of packs (and in the case of POPL12, a set of invariants), and constructs a DFG representation of the program. There are currently two strategies for constructing DFGs that are implemented in DUET, which we describe below.

Nested locks (SAS10) In [4], we leverage reachability results for concurrent programs communicating via nested locks [6] to develop a compositional technique for solving bitvector analysis problems. Data-dependence analysis can be formulated as a bitvector problem, so we may use this technique to construct

[1] `http://www.cprover.org/boolean-programs/`

DFGs. Our method computes a summary for each thread describing its behaviour and then composes the summaries to compute a DFG for the program. This compositional approach enables our analysis to be sound and precise (it computes meet-over-feasible-paths solutions) even when the number of executing threads in unbounded.

Global variables (POPL12) A more challenging (and also strictly more general) program model uses global variables for synchronization rather than locks. This model is difficult because it requires circular reasoning: in order to perform a reasonably accurate data dependence analysis to construct a DFG we need invariants for the synchronization variables, and in order to compute invariants for the synchronization variables we need to construct a DFG.

In [5], we present an approach based placing the DFG construction module (**POPL12**) and the **fixpoint solver** into a feedback loop. The DFG and the invariants are iteratively coarsened as the algorithm progresses (that is, the invariants become weaker, and more edges are added to the DFG) until a fixpoint is reached. When a fixpoint is reached, the invariants overapproximate the dynamic behaviour of the program in question.

3.4 Fixpoint solver

The **fixpoint solver** module interprets a DFG over an abstract domain to yield program invariants. It accomplishes this by computing a weak topological order on the DFG and repeatedly evaluating DFG nodes over the chosen abstract domain until a fixpoint is reached, as in [3]. DUET employs the abstract domains implemented in APRON [2] for this task.

3.5 Assertion checker

Finally, the **assertion checker** module iterates over the DFG vertices: for each assertion vertex, we determine whether the invariant at that location (computed by the fixpoint solver) implies the assertion using the APRON [2] library. If the check fails, the assertion *may* fail, and we add it to the error report.

4 Experiments

We used a benchmark suite of 15 Linux device drivers to evaluate DUET. Since a driver may have arbitrarily many clients, these programs exhibit unbounded parallelism. Table 1 presents the result of running DUET on a collection of 15 Linux device drivers. These drivers are all written in C, and include infinite data (such as integer types).

With an interval analysis, DUET manages to prove most of the assertions correct (1312 out of a total 1597), and does so in 13 minutes. DUET's performance using an octagon analysis is slightly worse, proving 1277 assertions correct in 90 minutes.

Most false positives for DUET appear to be caused by one of two reasons: imprecision in the abstract domain (e.g. lack of a precise enough abstraction to handle zero-terminated arrays that are used in most of these drivers), and

Device Drivers	#assertions	DUET: Interval Analysis		DUET: Octagon Analysis	
		safe	time	safe	time
`i8xx_tco`	90	75	1m51s	71	1m25s
`ib700wdt`	75	64	30s	64	20s
`machzwd`	87	73	39s	67	14m44s
`mixcomwd`	91	72	22s	74	25
`pcwd`	240	147	2m43s	145	23m48s
`pcwd_pci`	204	187	2m18s	188	2m59s
`sbc60xxwdt`	91	77	28s	69	11m27s
`sc520_wdt`	85	71	28s	65	13m20s
`sc1200wdt`	77	66	34s	66	33s
`smsc37b787_wdt`	93	80	47s	80	47s
`w83877f_wdt`	92	78	29s	72	13m24s
`w83977f_wdt`	101	90	34s	82	34s
`wdt`	99	88	25s	86	25s
`wdt977`	88	77	27s	75	28s
`wdt_pci`	84	67	33s	66	5m33s
total	1597	1312	13m9s	1270	90m21s

Table 1. DUET's Performance on Integer Programs with unbounded parallelism, run on an 3.16GHz Intel(R) Core 2(TM) machine with 4GB of RAM.

imprecision in how DUET handles the treatment of spinlocks in goto programs (due to the imprecision in the alias analysis that for lock variables). Neither of these sources of imprecision is due to a fundamental limitation of the analysis technique proposed in this paper (or related to concurrency). But, they hint on the idea that more precise alias analysis techniques (for concurrent code) and better abstract domains could hugely benefit the false positive rate of DUET.

		OCT	
		safe	unsafe
IVL	safe	1267	45
	unsafe	3	282

Interval vs Octagon Analysis The table on the right compares the results of interval and octagon analyses in DUET over the driver benchmarks (same experiments as in Table 1. Octagon analysis performs worse than interval analysis, however, it manages to prove 3 assertions safe that interval analysis fails. Octagon analysis is very sensitive to the variable packing algorithm. Investigating more sophisticated packing algorithms is a topic of our future work.

4.1 Boolean Programs

Although Boolean programs are not the intended target of DUET, we performed a set of experiments over existing Boolean program benchmarks for two reasons: (1) to compare with two recent approaches [8,7] for verification of concurrent Boolean programs with unboundedly many threads, and (2) there is no aliasing present in Boolean programs, which limits the scope of implementation-related imprecision for a better evaluation of the core method.

Even though DUET can directly analyze the original device driver codes (i.e. does not require a predicate abstraction phase), we chose to compare with the existing tools [8,7] on the Boolean abstractions (for which these tools were designed), to present a more fair comparison. We compare DUET against two recent algorithms that handle Boolean programs with unbounded parallelism: dynamic cutoff detection (DCD) from [7], as implemented in Boom, and linear interfaces (LI) from [8], as implemented in Getafix. We compared these tools against the benchmarks provided by the authors. It is important to note that both tools are capable of finding counterexamples to definitively declare an assertion unsafe, whereas, when Duet fails to produce strong enough invariants to prove an assertion correct, it is not clear whether the assertion is incorrect or Duet failed to prove it correct. The timeout is 5 minutes in all cases.

		LI			
		safe	unsafe	timeout	unknown
Duet	safe	1289	0	267	957
	unknown	54	247	23	579
	timeout	0	0	0	0

The table on the left presents the results of comparing Duet against LI over LI benchmarks. Columns/rows titled safe, unsafe, and timeout are self-explanatory. An "unknown" response from Duet means that the invariants were not strong enough to prove the assertion safe, and an "unknown" response from LI means that neither a counter example was found nor the program was proved safe. Duet substantially outperforms LI; for example, there were 957 assertions proved safe by Duet that were declared unknown by LI, where as LI could only prove safe 54 of Duet's unknown cases.

The table on the right presents the results of comparing against DCD over DCD benchmarks. Again, Duet substantially outperforms DCD in proving assertions correct; 39 more assertions (out of 58) are proved correct by Duet.

		DCD		
		safe	unsafe	timeout
Duet	safe	19	0	39
	unknown	0	203	2
	timeout	0	7	0

References

1. J. Alglave, D. Kroening, N. He, A. Ranjan, N. Seghir, and M. Tautschnig. CPROVER project, Nov. 2011.
2. J. Bertrand and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications*, pages 128–141, 1993.
4. A. Farzan and Z. Kincaid. Compositional bitvector analysis for concurrent programs with nested locks. In *SAS*, pages 253–270, 2010.
5. A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, pages 297–308, 2012.
6. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.
7. A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, pages 645–659. 2010.
8. S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, pages 629–644. 2010.
9. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, pages 213–228, 2002.