
# Verified Data Structures for Trusted Autonomy: A Compilation Approach

David Hardin
Konrad Slind

Rockwell Collins

# Motivation

- Verification and Validation of Autonomous Systems is a significant issue throughout DoD
- A commonly-held view is that current testing-based V&V regimes are inadequate
  - A human operator has long been relied upon as the ultimate "safety monitor" — which truly autonomous systems lack
- Unfortunately, the sophistication of autonomous systems development techniques makes V&V even more difficult
  - "Deep learning" approaches thwart traditional requirements-driven, test-coverage-driven V&V, making it difficult to even provide a straightforward explanation of any given decision
    - We're not tackling this problem here!
  - Even basic machine reasoning for inference, route planning, etc. present a significant V&V challenge, due to their use of complex data types and subtle algorithms

# Motivation (cont'd.)

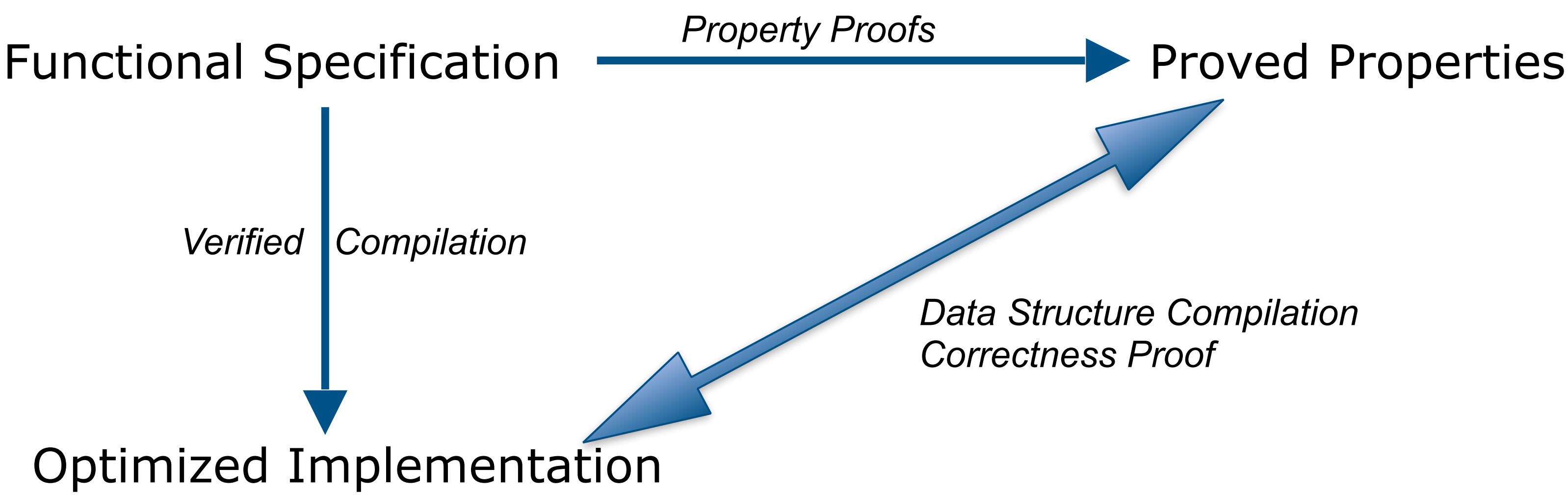
- Autonomy algorithms, e.g. route planning, employ complex algebraic data types
- Proof techniques for these data structures exist, but are oriented to unbounded, functional data types
  - Functional data structure implementations are not often efficient in space or time, so developers generally take a more imperative approach
- We need to find proof techniques that embrace the "natural" functional proof style, yet apply to more efficient data structure implementations
  - Including GPU-based and hardware-based data structures

# Our Approach: Verified Data Structure Compilation to Linearized Form

- Accepts Data Structure Specification from parsed ML-like data structure specification
  - Data structure specification includes a maximum size
- Compiles the Data Structure Specification into a linearized form requiring no heap allocation or deallocation, in keeping with high-assurance development tenets (e.g. DO-178C Level A)
  - Allocation/deallocation may be added later for systems that need it
- Produces proofs that compiled data structure operations on the compiled form are equivalent to the same operations on the functional form
  - Proves that in-place updates are equivalent to functional (copying) updates, given that no "old" copies of the data structure are allowed

# Verified Data Structure Compilation and Property Proofs

- Once we develop the Data Structure Compilation Correctness Proof, properties proved of the functional data structure specification will also hold for the optimized implementation



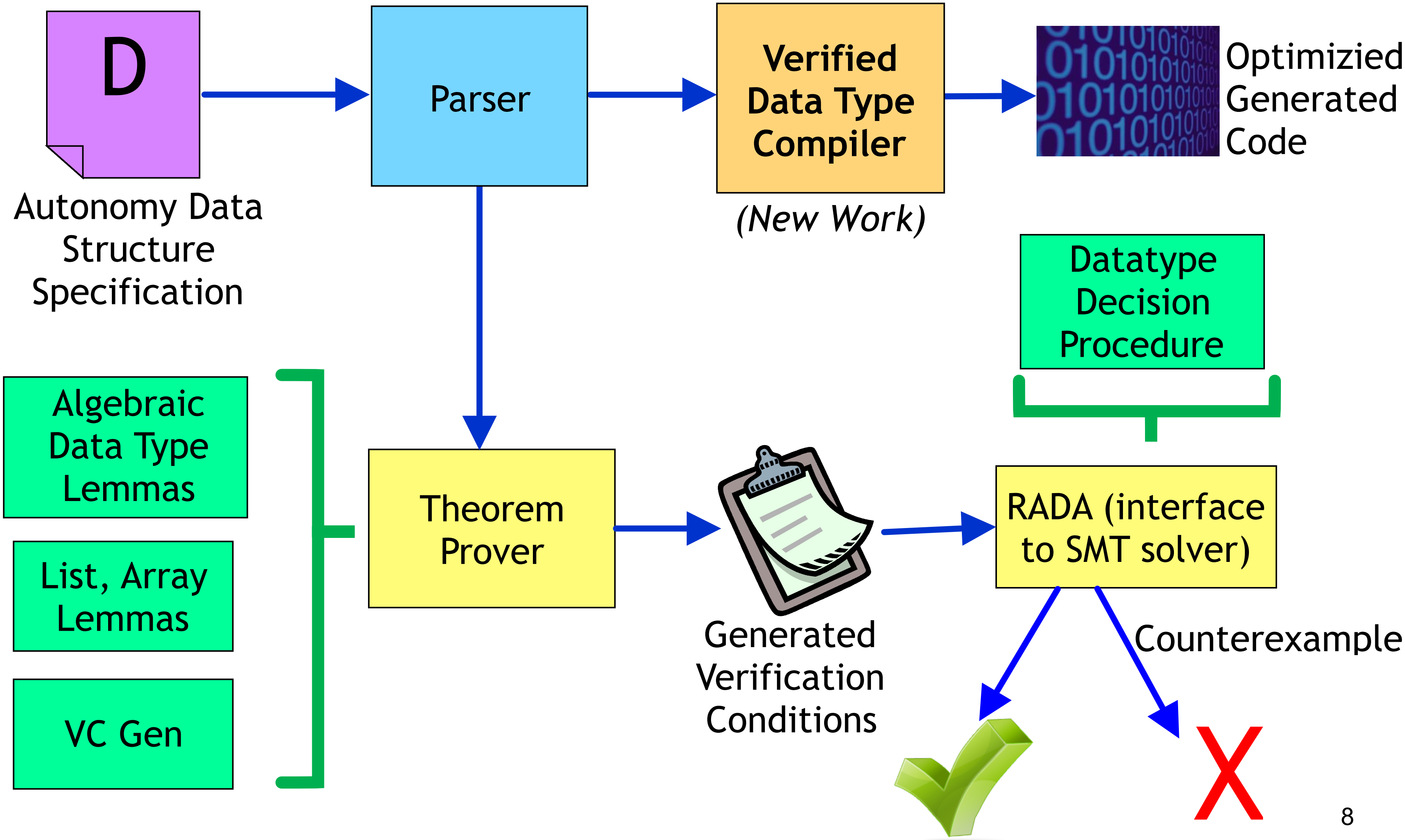
# Touchstones for our Work

- Experience on Autonomy programs, e.g. AFRL Loyal Wingman
- DO-178C Airborne Systems Certification Standard (RTCA 2012)
- Guardol DSL for Cross-Domain Systems (TACAS 2012)
- Guardol Verified Compilation to VHDL (SAFECOMP 2016)
- Accelerating Large Graph Algorithms on the GPU using CUDA (Harish and Narayanan, HiPC 2007)
- ACL2 Single-Thread Objects; functional programs with imperative implementations (Boyer and Moore, PADL 2002)
- Formalization of a CUDA-based Parallelizable All-Pairs Shortest Path Algorithm in ACL2 (ACL2 Workshop 2013)
- Decompilation into Logic (Myreen, Dissertation 2009)
- Verification-Enhanced Languages (Dafny, SPARK, Guardol)
- Verified Compilers (CompCert, CakeML)
- MASC: SystemC in ACL2 (O'Leary and Russinoff 2014)

# The ACL2 Single-Threaded Object (stobj)

- The ACL2 theorem prover provides a declaration mechanism to create so-called “single-threaded objects”, or stobjs
- ACL2 enforces strict syntactic rules on stobjs to ensure that “old” states of a stobj are guaranteed not to exist
  - This means that ACL2 can provide destructive implementation for stobjs, allowing stobj operations to execute quickly
- An ACL2 single-threaded object thus combines:
  - a functional semantics about which we can reason
  - a relatively high-speed implementation that more closely follows “normal” design rules for high assurance
- ACL2 stobjs have been used to produce, e.g. a high-speed, detailed operational semantics for x86-64 that can process up to 3 million simulated x86-64 instructions per second
- We make extensive use of the stobj *idea* in our DASL compiler

# Autonomy Data Type Verification Overview



# From Domain-Specific to Domain-Aware Programming Languages

- Our work on Guardol, a Domain-Specific Language for cross-domain systems led us to the realization that it is often more useful for a language to be *domain-aware* than *domain-specific*

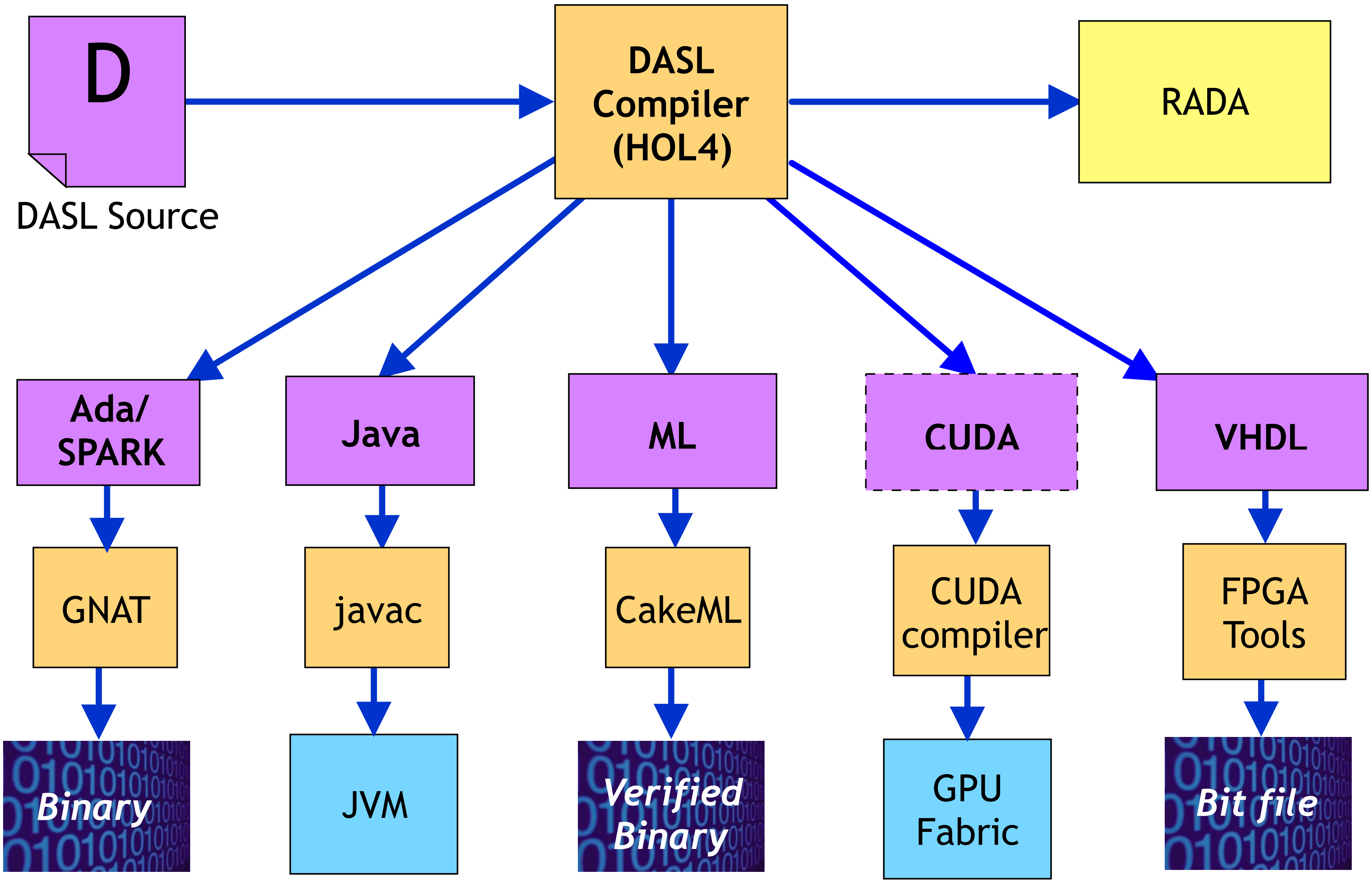
- Further, domain-aware language design principles can apply to a number of domains with similar computational, environmental, and regulatory requirements (e.g., embedded safety-critical domains, security-critical domains)

- Thus, we have created a Domain-Aware Programming Language (DAPL) for autonomy applications, called DASL

# The Domain-Aware System Language (DASL)

- DASL is designed for the creation of efficient, verifiable, accreditable algorithms in domains such as autonomy
- DASL is a system-level language, appropriate for expressing algorithms and data structures that can be compiled to traditional programming languages, GPU languages, as well as Hardware Description Languages (HDLs)
- DASL can be characterized as a "mashup" of concepts from Ada, ML, and the C family of languages, and has a similar feel to new languages such as Swift and Rust
- The DASL toolchain is designed to support Formal Verification, and utilizes the HOL4 theorem prover as its "middle end"

# DASL Code Generation Options



# The DASL Language

- DASL is, similar to Guadol, a strongly-typed imperative language, with assignment, functions, for loops, while loops, etc., but with data types influenced by ML:

```
datatype Tree = [elem: int, rank: int, children TreeList];

datatype TreeList
  = Nil
  | Cons : [hd: Tree, tl: TreeList]}

type IntOpt = { NONE | SOME : int }

function ins (t: in Tree, tlist: in TreeList) returns Ret: TreeList {
  match tlist {
    'Nil => Ret := 'Cons [hd: t, tl: 'Nil];
    'Cons c =>
        if t.rank < c.hd.rank then
          Ret := 'Cons [hd: t, tl: tlist];
        else
          Ret := ins (link (t, c.hd), c.tl); }}
```

# DASL Language and Toolchain Attributes for Formal Analysis

- DASL does not allow arbitrary pointers or explicit pointer arithmetic
- Arrays are of fixed size, and array accesses are subject to mandatory bounds-checking
- DASL does not support goto or setjmp/longjmp
- HOL4 gives semantics to DASL evaluation via decompilation into logic, and we use proved source-to-source transformations in HOL4 to compile DASL code
- Technique from (Greve and Slind 2013) allows DASL functions to be introduced into the logic with deferred termination proofs
- **`spec`** statements: Allow user to write property specifications in DASL syntax that can be proven by the DASL backend

## DASL Property Specifications and Proofs

- A feature of DASL inherited from Guardol is the ability to state and prove formal property specifications directly in the source text, using DASL language syntax
- The following property spec conjectures that if a TreeList is rank-ordered, it is still rank-ordered after a new tree is inserted:

```
spec rank_ordered_ins = {
   var t: Tree;
       list: TreeList;
   in
     if rank_ordered(list)
       then check rank_ordered(ins(t, tlist));
     else skip;
 }
```

- The DASL verification backend proves this property automatically

## The `sized` Declarator and Compilation to Array-Based Form

- A new feature of DASL is the **`sized`** declarator, which informs the toolchain that an otherwise unbounded datatype declaration has limited size:

```
sized pq: PQType (MAX_VERTICES);
```

- sized datatypes can be compiled to an array-based form with destructive updates, similar to the way that ACL2 stobjs are compiled

- Array-based form greatly simplifies code generation for GPUs and hardware

# DASL Graph Datatypes

- Another new feature of DASL is a specialized graph datatype declarator, and its associated sized declarator:

```
graphtype DKGraph (nodeLabel = vertexLabelTy,
                   edgeLabel = edgeLabelTy);

 sized dkg: DKGraph (MAX_VERTICES, MAX_EDGES_PER_VERTEX);
```

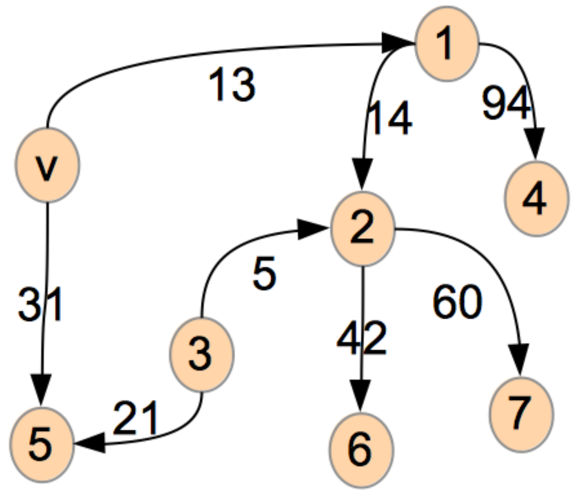
- The DASL toolchain compiles this declaration to an array-based form, and generates several associated functions for manipulating the array-based form:

```
getOutEdges(), setOutEdges(), addEdge(), labelVertex(),
 labelEdge(),…
```

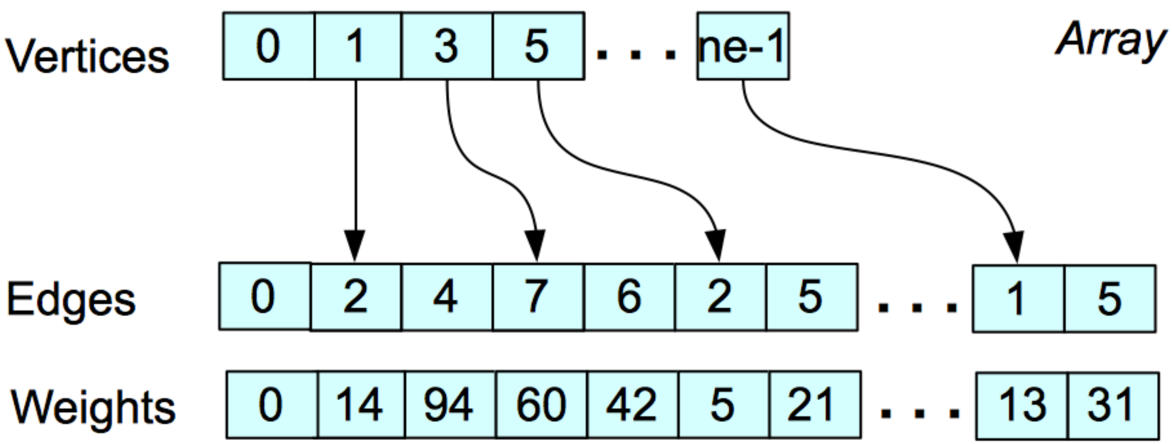
# Array-Based Graph Representation

- Based on a data structure layout approach created for efficient GPU execution (Harish and Narayanan, HiPC 2007); used to code Dijkstra’s All-Pairs Shortest Path algorithm (APSP)
- Amenable to efficient CUDA, OpenCL implementation, as well as hardware implmentation (VHDL)
- Implementated APSP using ACL2 single-threaded object (ACL2 Workshop 2013)
  - Execution of Dijkstra’s shortest path algorithm on compiled graph using stobjs was linear in number of vertices up to at least 1 million vertices at 10 edges per vertex
- DASL compiler analyzes **`datatype`**, **`graphtype`**, and **`sized`** declarations, creates appropriate array-based layout, and instantiates runtime functions

# Graph Compilation Example, Two Edges per Vertex




*Graph (incomplete)*




*Array representation*

# Research Results to Date

- Defined DASL language features, focusing on **`sized`** data structures
- Defined low-level data structure layouts similar to those used in ACL2-13 paper, starting with prototypes in ACL2
- Defined "runtime" functions generated by the toolchain to operate on the low-level data structures, using ACL2
  - Performed basic correctness proofs of runtime functions
- Updated HOL4-based toolchain to support **`datatype`**, **`graphtype`** declarations, and providing sized data structure compilation correctness proofs
- Wrote DASL programs for a number of autonomy-relevant algorithms and datastructures: tree search, priority queue, Dijkstra APSP, unify/substitute
- Generating Ada code for **`datatype`**, **`graphtype`** declarations

## Next Steps

- Complete working end-to-end examples for Dijkstra APSP, A*, unify, etc. including high-level property proofs
    - Proofs will utilize a combination of HOL4 and RADA
- Refine language and runtime definitions based on our experience
- Port runtime to Java, ML, CUDA, VHDL
- Make the generated Ada code SPARK-conformant
- Develop CUDA code generation
- Gain experience on larger examples, esp. in collaboration with the Flex project at Kestrel