

O'REILLY®  
**OSCON**  
open source convention



oscon.com



# Introduction to ParaSail

*Parallel Specification and Implementation Language*

S. Tucker Taft  
SofCheck, Inc.  
OSCON July 2011





## Outline of Presentation

- Why Design A New Language for High-Integrity Systems?
- Simplify, Unify, Parallelize, Formalize
- How does ParaSail compare?
- Conclusions

# Why Design A New Language for High-Integrity Systems?

- 80+% of safety-critical systems are developed in C and C++, two of the least safe languages invented in the last 40 years
- Every 40 years you should start from scratch
- Computers have stopped getting faster ----->
- By 2020, most chips will have 50-100 cores
- Advanced Static Analysis has come of age
  - time to get the benefit at compile-time

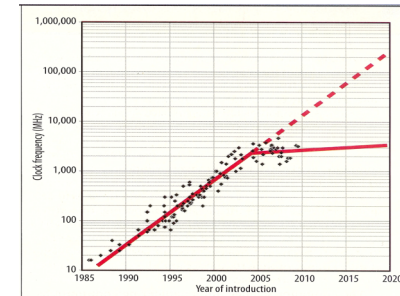
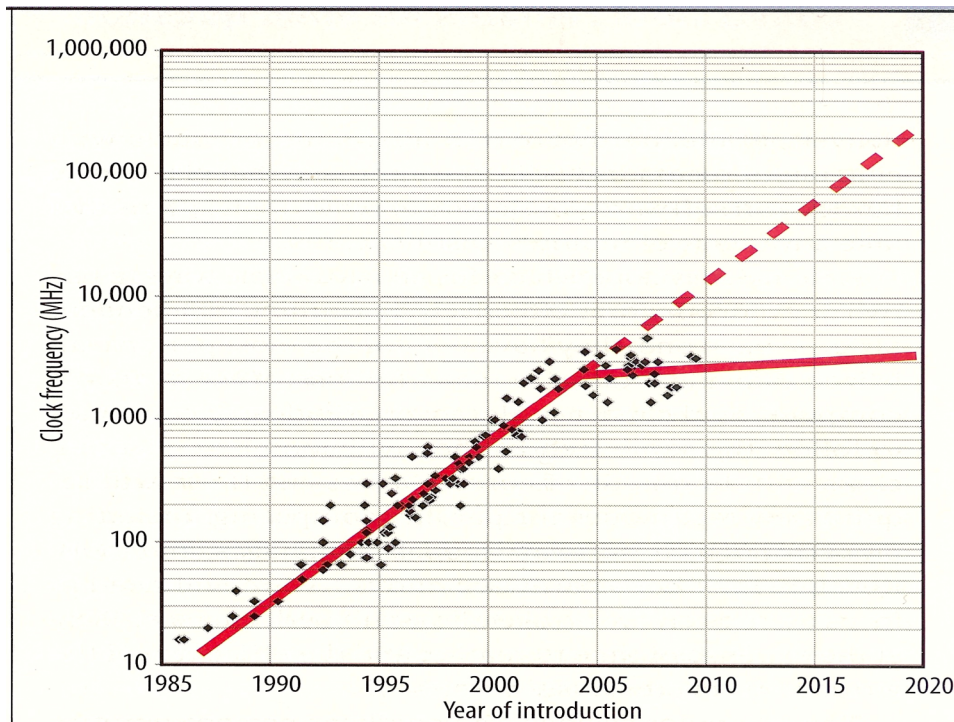


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

# Why a new language?

## Computers have stopped getting faster



**Figure 2.** Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Courtesy IEEE  
Computer,  
January 2011,  
page 33.



# Building on Existing Languages

- Lisp, ML, and friends
  - Lisp, CLOS, Scheme, ML, OCaml, F#, Scala, Clojure
- More conventional languages
  - Modula, Oberon, Ada, Cyclone, C++, Java, C#, Python
- Parallel programming languages
  - C\*, HPF, CUDA, Cilk, OpenCL, CnC
  - Challenges both in coding and then debugging
- Languages with assertions, pre/postconditions, etc.
  - Eiffel, SPARK, Ada 2012, JML, SAL
  - Both for specification and for debugging
  - Manage the growing complexity and parallelism

# Simplify, Unify, Parallelize, Formalize

## ■ Simplify/Unify

- Smaller number of concepts, uniformly applied, all features available to user-defined types
- Simplify to make *conceptual room* for parallelism and formalism

## ■ Parallelize

- Parallel by default
- Have to work harder to force sequential execution

## ■ Formalize

- Assertions, Invariants, Preconditions, Postconditions integrated into the syntax
- All checking at compile-time -- if compiler can't prove the assertion, program is illegal
- No run-time checking, no run-time exceptions



# Simplified Module/Type/Object Model



- ParaSail has four basic concepts:
  - Module
    - has an Interface, and Classes that implement it
    - **interface** Set <Element\_Type is Comparable<>> **is** ...
    - Supports *inheritance* of interface and code
  - Type
    - is an instance of a Module
    - **type** T **is** Set <Integer>;
    - “T+” is *polymorphic* type for types inheriting from T’s interface
  - Object
    - is an instance of a Type
    - **var** Obj : T := T::Create(...);
  - Operation
    - is defined in a Module, and
    - operates on one or more Objects of specified Types.

## Example: N-Queens Interface



```
interface N_Queens <N : Univ_Integer := 8> is  
  // Place N queens on an NxN checkerboard so that none of them can  
  // "take" each other. Return vector of solutions, each solution being  
  // an array of columns indexed by row indicating placement of queens.  
  
  type Chess_Unit is new Integer<-N*2 .. N*2>;  
  type Row is Chess_Unit {Row in 1..N};  
  type Column is Chess_Unit {Column in 1..N};  
  type Solution is Array<optional Column, Indexed_By => Row>;  
  
  func Place_Queens() -> Vector<Solution>  
    {for all S of Place_Queens => for all C of S => C not null};  
end interface N_Queens;
```



© 2011 SofCheck, Inc.

O'REILLY  
**OScon**  
open source convention



## Simplify/Unify Arrays/Containers

- Collections/Containers: Array, Map/Hashtable, Tree, Set, Vector, Linked list, Sequence, ...
  - Elements are “key => value” or “key => is\_present”
  - Homogeneous (at compile-time)
    - might be polymorphic at run-time (via a tag of some sort)
  - Iterators, indexing, slicing, combining/merging/concatenating
  - Empty container representation (e.g. “[]”)
  - Explicit “literal” instance, e.g.:
    - [2|3|5|7 => #prime, .. => #composite]
  - May grow or shrink over time



## ParaSail Approach for Containers

- `Container[Index]` for indexing
- `Container[A..B]` for slicing
- `[]` for empty container
- `[key1..key2=>val1, key3=>val3]` or `[val1, val1, val3]` for literal container aggregate
- `X|Y` for combining/concatenating/merging
- `C|=Y` or `C|=[key=>Y]` for adding Y to container C
- *User* defines operators “indexing”, “[]”, and “|=” and then compiler will create temps to support “X | Y” and “[...]” aggregates.

## Containers Instead of Pointers

- Generalized *indexing* into containers replaces pointer dereferences
  - Similar to region-based storage management
  - `Region[Index]` equiv to `*Index`
    - presuming `Index` points into `Region`
- Objects can be declared **optional** and/or resizable (**mutable**)
  - Eliminates many of the common uses for pointers
- Short-lived *references* to existing objects are permitted
  - Returned by user-defined indexing functions, for example

# User-defined Indexing, Literals, etc.

## ■ User-defined *indexing*

- Any type with **operator** “indexing” defined
- Indexing function returns **ref** to component of parameter
- Built-in support for extensible structures, optional elements

## ■ User-defined *literals*

- Any type with **operator** “from\_univ” defined from:
  - Univ\_Integer (42), Univ\_Real (3.141592653589793)
  - Univ\_String (“Hitchhiker’s Guide”), Univ\_Character (‘π’)
  - Univ\_Enumeration (#red)

## ■ User-defined *ordering*

- Define single binary **operator** “=?” (pronounced “*compare*”)
- Returns #less, #equal, #greater, #unordered
- Implies “<=”, “<”, “==”, “!=”, “>”, “>=”, “in X..Y”, “not in X..Y”

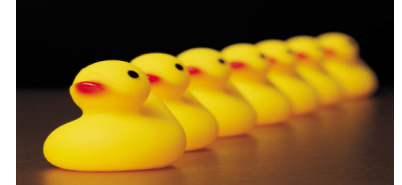




## Why and How to Parallelize?

- Computers have *stopped* getting faster -- they are getting “*fatter*” -- more cores, not more GHz
- Programmers are *lazy* -- will take path of least resistance
  - ⇒ The *default* should be *parallel* -- must work harder to force sequential evaluation.
  - ⇒ Programmer mindset: *there are 1000s of threads, and the goal is to use as many as possible.*
- Compiler should *prevent* race conditions, and ideally, deadlock as well.

## Parallelism in ParaSail



- *Parallel* by default
  - *expressions* are evaluated in *parallel*
  - have to work *harder* to make code run *sequentially*
- Easy to create even *more* parallelism
  - Process(X) || Process(Y) || Process(Z);
  - for I in 1..10 **concurrent** loop ... end loop;
- *Lock-based* and *lock-free* **concurrent** objects
  - Lock-based objects also support conditionally **queued** access
  - User-defined *delay* and *timed* call based on **queued** access
- No *global* variables
  - Operation can *only* access or update variable state via its *parameters*
- Compiler prevents *aliasing* and *unsafe access* to non-concurrent variables
  - No *pointers*; no *sharing* of data between objects.
  - Overall *pure value semantics* for non-concurrent objects

## Expression and Statement Parallelism

- *Within* an expression, parameters to an operation are evaluated in parallel
  - $F(X) + G(Y) * H(Z)$  --  $F(X), G(Y), H(Z)$  evaluated concurrently
- Programmer can force parallelism *across* statements
  - $P(X) \parallel Q(Y)$
- Programmer can force sequentiality
  - $P(X) \text{ then } Q(Y)$
- *Default* is run in *parallel* if no dependences
  - $A := P(X); B := Q(Y)$  -- can run in parallel
  - $Y := P(X); B := Q(Y)$  -- cannot run in parallel

## More Examples of ParaSail Parallelism

```
for X => Root then X.Left || X.Right while X not null
  concurrent loop
    Process(X.Data);    // Process called on each node in parallel
end loop;
```

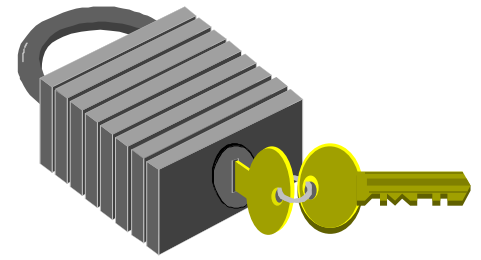
---

```
concurrent interface Box<Element is Assignable<>> is
  func Create() -> Box;    // Creates an empty box
  func Put(locked var B : Box; E : Element);
  func Get(queued var B : Box) -> Element; // May wait
  func Current_Contents(locked B : Box) -> optional Element;
end interface Box;
type Item_Box is Box<Item>;
var My_Box : Item_Box := Create();
```



# Synchronizing ParaSail Parallelism

```
concurrent class Box <Element is Assignable<>> is  
  var Content : optional mutable Element; // starts null and can change size  
exports  
  func Create() -> Box is // Creates an empty box  
    return (Content => null);  
  end func Create;  
  func Put(locked var B : Box; E : Element) is  
    B.Content := E;  
  end func Put;  
  func Get(queued var B : Box) -> Element is // May wait  
    queued until B.Content not null then  
      const Result := B.Content;  
      B.Content := null;  
      return Result;  
    end func Get;  
  func Current_Contents(locked B : Box) -> optional Element is  
    return B.Content;  
  end func Current_Contents;  
end class Box;
```



## ParaSail Virtual Machine

- ParaSail Virtual Machine (PSVM) designed for prototype implementations of ParaSail.
- PSVM designed to support “pico” threading with *parallel block*, *parallel call*, and *parallel wait* instructions.
- Heavier-weight “server” threads serve a LIFO queue of light-weight *pico-threads*, each of which represents a sequence of PSVM instructions (*parallel block*) or a single *parallel “call”*
  - Similar to Intel’s **Cilk** run-time model with *work stealing*.
- While waiting to be served, a *pico-thread* needs only a *handful* of words of memory.
- A single ParaSail program can easily involve *1000’s* of pico threads.
- PSVM *instrumented* to show degree of parallelism achieved

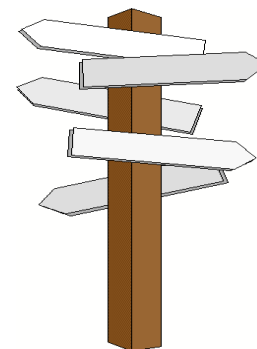


## Why and How to Formalize?

- *Assertions* help catch bugs sooner rather than later.
- Parallelism makes bugs much more *expensive* to find and fix.
- ⇒ *Integrate* assertions (annotations) into the syntax everywhere, as pre/postconditions, invariants, etc.
- ⇒ Compiler *disallows* potential race-conditions.
- ⇒ Compiler *checks* assertions, rejects the program if it can't prove the assertions.
- ⇒ *No* run-time checking implies better performance, and *no* run-time exceptions to worry about.

## Annotations in ParaSail

- Preconditions, Postconditions, Constraints, etc.
  - all use same Hoare-like syntax:  $\{X \neq 0\}$
- All assertions are checked at compile-time
  - no run-time checks inserted
  - no run-time exceptions to worry about or propagate
- Location of assertion determines whether is a:
  - precondition (before “->”)
  - postcondition (after “->”)
  - assertion (between statements)
  - constraint (in type definition)
  - invariant (at top-level of class definition)





## Examples of ParaSail Annotations

```
interface Stack <Component is Assignable<>; Size_Type is Integer<>> is
  func Max_Stack_Size(S : Stack) -> Size_Type {Max_Stack_Size > 0};

  func Count(S : Stack) -> Size_Type
    {Count <= Max_Stack_Size(S)};

  func Create(Max : Size_Type {Max > 0}) -> Stack
    {Max_Stack_Size(Create) == Max and Count(Create) == 0};

  func Is_Empty(S : Stack) -> Boolean
    {Is_Empty == (Count(S) == 0)};

  func Is_Full(S : Stack) -> Boolean
    {Is_Full == (Count(S) == Max_Stack_Size(S))};

  func Push(var S : Stack {not Is_Full(S)}; X : Component)
    {Count(S') == Count(S) + 1};

  func Top(ref S : Stack {not Is_Empty(S)}) -> ref Component;

  func Pop(var S : Stack {not Is_Empty(S)})
    {Count(S') == Count(S) - 1};
end interface Stack;
```

## More on Stack Annotations

```
class Stack <Component is Assignable<>; Size_Type is Integer<>> is
  const Max_Len : Size_Type;
  var Cur_Len : Size_Type {Cur_Len in 0..Max_Len};
  type Index_Type is Size_Type {Index_Type in 1..Max_Len};
  var Data : Array<optional Component, Indexed_By => Index_Type>;
exports
  {for all I in 1..Cur_Len => Data[I] not null}    // invariant for Top()
  func Count(S : Stack) -> Size_Type
    {Count <= Max_Stack_Size(S)} is
    return S.Cur_Len;
  end func Count;
  func Create(Max : Size_Type {Max > 0}) -> Stack
    {Max_Stack_Size(Create) == Max and Count(Create) == 0} is
    return (Max_Len => Max, Cur_Len => 0, Data => [.. => null]);
  end func Create;
  func Push(var S : Stack {not Is_Full(S)}; X : Component)
    {Count(S') == Count(S) + 1} is
    S.Cur_Len += 1;           // requires not Is_Full(S) precondition
    S.Data[S.Cur_Len] := X;   // preserves invariant (see above)
  end func Push;
  func Top(ref S : Stack {not Is_Empty(S)}) -> ref Component is
    return S.Data[S.Cur_Len]; // requires invariant (above) and not Is_Empty
  end func Top; ...
end class Stack;
```

## More Parasail Examples



# Walk Parse Tree in Parallel

```
type Node_Kind is Enum < [#leaf, #unary, #binary] >;
...
for X => Root while X not null loop
  case X.Kind of
    [#leaf] =>
      Process_Leaf(X);
    [#unary] =>
      Process_Unary(X.Data) ||
      continue loop with X => X.Right;
    [#binary] =>
      Process_Binary(X.Data) ||
      continue loop with X => X.Left ||
      continue loop with X => X.Right;
  end case;
end loop;
```



## Parallel N-Queens Interface



```
interface N_Queens <N : Univ_Integer := 8> is  
  // Place N queens on an NxN checkerboard so that none of them can  
  // "take" each other. Return vector of solutions, each solution being  
  // an array of columns indexed by row indicating placement of queens.  
  
  type Chess_Unit is new Integer<-N*2 .. N*2>;  
  type Row is Chess_Unit {Row in 1..N};  
  type Column is Chess_Unit {Column in 1..N};  
  type Solution is Array<optional Column, Indexed_By => Row>;  
  
  func Place_Queens() -> Vector<Solution>  
    {for all S of Place_Queens => for all C of S => C not null};  
end interface N_Queens;
```

# Parallel N-Queens Class

## (cont'd)



```
class N_Queens is
  type Sum_Range is Chess_Unit {Sum_Range in 2..2*N};
  type Diff_Range is Chess_Unit {Diff_Range in (1-N) .. (N-1)};
  type Sum is Set<Sum_Range>;
  type Diff is Set<Diff_Range>;
  exports
    func Place_Queens() -> Vector<Solution>
      {for all S of Place_Queens => for all C of S => C not null}
  is
    var Solutions : concurrent Vector<Solution> := [];
    *Outer_Loop*
    for (C : Column := 1; Trial : Solution := [.. => null];
        Diag_Sum : Sum := []; Diag_Diff : Diff := []) loop
      // Iterate over the columns
      ...
      // All done, remember trial result with last queen placed
      Solutions |= (Trial | [R => C]);
      ...
    end loop Outer_Loop;
    return Solutions;
  end func Place_Queens;
end class N_Queens;
```



## Parallel N-Queens Class (cont'd)



```
func Place_Queens() -> Vector<Solution> is
  var Solutions : concurrent Vector<Solution> := [];
  *Outer_Loop*
  for (C : Column := 1; Trial : Solution := [.. => null];
    Diag_Sum : Sum := []; Diag_Diff : Diff := []) loop // over the columns
    for R in Row concurrent loop // over the rows
      if Trial[R] is null and then
        (R+C) not in Diag_Sum and then (R-C) not in Diag_Diff then
          // Found a Row/Column combination that is not on any diagonal
          if C < N then // Keep going since haven't reached Nth column.
            continue loop Outer_Loop with (C => C+1,
              Trial => Trial | [R => C],
              Diag_Sum => Diag_Sum | (R+C),
              Diag_Diff => Diag_Diff | (R-C));
          else // All done, remember trial result with last queen placed
            Solutions |= (Trial | [R => C]);
          end if;
        end if;
      end loop;
    end loop Outer_Loop;
  return Solutions;
end func Place_Queens;
```

# What makes ParaSail Interesting?



- Pervasive (implicit and explicit) *parallelism*
  - Supported by ParaSail Virtual Machine
- Inherently *safe*:
  - preconditions, postconditions, constraints, etc., integrated throughout the syntax
  - no global variables; no dangling references; value semantics
  - no run-time checks or exceptions -- all checking at compile-time
  - storage management based on optional and extensible objects
- Small number of flexible *concepts*:
  - Modules, Types, Objects, Operations
- *User-defined* literals, indexing, aggregates, physical units checking
- It's got a *cool* name

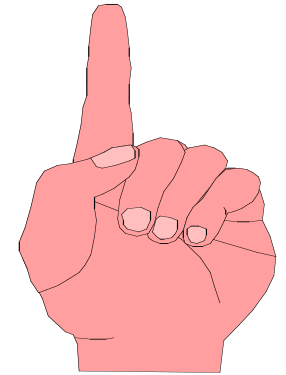




## How does ParaSail Compare to ...

- *C/C++* -- built-in safety; built-in parallelism; much simpler
- *Java* -- eliminates race conditions, increases parallelism, avoids garbage collection, does all checking at compile-time, no run-time exceptions
- *Ada* -- eliminates race conditions, increases parallelism, eliminates run-time checks, simplifies language
- *OCaml/F#* -- unifies modules and objects, eliminates exceptions, avoids garbage collection, increases parallelism, more emphasis on readability
- *Haskell* -- provides many of the benefits of pure functional languages (e.g. no aliasing), but more familiar model; less need for compiler to transform for efficiency

## Conclusions and Ongoing Work



- It is productive to start from scratch now and then
- Can simplify and unify
- Can focus on new issues
  - Pervasive *parallelism*
  - Integrated *annotations* enforced at *compile-time*
- Ongoing work
  - Completing Prototype Compiler and ParaSail Virtual machine
  - Draft *ParaSail Reference Manual* now available
  - Giving presentations and tutorials on ParaSail
- Read the *blog* if you are interested...
- <http://parasail-programming-language.blogspot.com>



© 2011 SofCheck, Inc.

O'REILLY  
**OSCON**  
open source convention

O'REILLY®

**OSCON**  
open source convention



oscon.com



11 Cypress Drive  
Burlington, MA 01803

**Tucker Taft**

tucker.taft@sofcheck.com

<http://parasail-programming-language.blogspot.com>

+1 (781) 750-8068 x220

