# Parallel Support for Source Code Analysis and Modification

Ira D. Baxter
Semantic Designs, Inc.
idbaxter@semdesigns.com

## Abstract

*Tools that analyze and enhance large-scale software systems using symbolic reasoning are computationally expensive, and yet processors are cheap. We believe that enabling tools with parallel foundations will lead to qualitatively more useful tools. We have implemented a large-scale industrial-strength program transformation system, the DMS Software Reengineering Toolkit, entirely in PARLANSE, a new parallel language. PARLANSE provides support for irregular fine-grain parallelism with zero-cost exception handling. This paper sketches the motivation for PARLANSE, its parallelism support and how it is used in a number of DMS-based applications, including parallel rewriting and attribute evaluation.*

## 1 Introduction: Symbolic Computation and Parallelism

Automation is the key to productivity gains in software engineering. Such automation seems best obtained by enabling tools to reason collectively and deeply about the problem domain concepts, software design and code of application software. While this leads to a number of scaling issues including knowledge modularization as *domains* [Neighbors84], long-term design transactions and incremental design modification [Baxter92], this paper focuses on alleviating the computational costs induced simply from processing large software systems, and computing symbolic results, by applying parallel computing methods for irregular computations.

Microprocessors are blazingly fast at integer arithmetic, but not nearly so good at symbolic computation. Symbolic computation for software engineering tasks are growing in scale as the software applications grow. This suggests a growing demand for cycles, and one method for alleviating that demand is parallel computation.

One doesn't simply retrofit an application with parallelism; this has proven to be a very difficult task in the scientific computation world, which generally has "simpler" computations than symbolic ones. When designing tools for large symbolic computations, one should then consider how parallelism will be built into such a tool from the very beginning. This in turn requires a careful analysis of what kinds of parallelism the tool should use and how that will be supported.

A method for supplying parallelism to an application is to build it on top of a conventional multiprocessing operating system that supplies a threads package, program the application in any convenient programming language such as C, and hand-code calls on the threads package when parallel opportunities are available. For complex parallelism tasks, this has proven to be impossible to code, debug and maintain, and is a sure route to failure for a large application. Worse, such applications tend to be very unrobust in the face of exceptions, because thread packages offer no organized way to manage exceptions in the face of complex parallel structures.

Handling complex entities by use of ("domain-specific") languages in which those entities are explicit is often the key to managing such entities. Parallelism, we claim, is no different; if one wants to manage complex parallelism well, then defining a language in which the concepts are explicit makes them easier to code and can enable a compiler to take care of all the complex interactions that ensue, allowing an application programmer to pursue his task more effectively.

We have built a industrial-strength program analysis and transformation tool called the "DMS Software Reengineering Toolkit", with the intention and currently some initial success of applying it commercially to very large software systems having multiple source languages, tens of thousands of source files, and millions of lines of code. We designed and implemented PARLANSE, a programming language with explicit support for fine-grain irregular parallelism with strong exception support, and built DMS entirely in PARLANSE. With several years of implementation and basic experience DMS and PARLANSE behind us, we are able to describe some interesting applications of PARLANSE, suggesting that we are on the right track.

Section 2 discusses the scaling problems using microprocessors for symbolic computation. Section 3 outlines DMS, an industrial-strength program transformation system that faces these issues by employing a parallel programming language, PARLANSE, as its foundation. Section 4 lists a number of industrial applications of DMS. Section 5 discusses irregular parallelism as the basis for parallel symbolic computation. Section 6 sketches PARLANSE and its support for parallelism. Section 7 covers a number of parallel symbolic applications to which PARLANSE has been applied. Section 8 provides a summary of results.

## 2 Microprocessors and Symbolic Computation

Modern microprocessors are designed to execute basic integer instructions quickly by using deep pipelines, multiple function units, “large” instruction and data caches, and branch prediction/avoidance schemes. Deep pipelines enable the execution of multiple instructions by overlapping execution phases; multiple functional units allow the harnessing of instruction-level parallelism. This can often achieve peak rates of 1-3 instructions per clock. Large caches (typically 250K-2Mb) lower the cost of repeated memory accesses to the same data. Typical data access at memory cycle times is about 50nS and caches typically reduce access time to effectively single-clock access times to previously captured data. Since caches are modest in size compared to the main memory, any program that must touch data volumes larger than the cache are reduced to memory access times rather than cache access times. A deep pipeline must be continually fed new instructions to maintain high throughput, and conditional branches make it difficult to determine which sequence of instructions to follow. Mispredicting branches can cause such deep pipelines to flush when a branch is taken, reducing the throughput rate to 1 instruction for the tens of clocks it takes to refill the pipe, losing a performance factor of 60 or more. Consequently, such processors implement many heuristics to predict branches, and even offer methods to avoid branches by defining instructions with conditional execution (Intel MOVCC, ARM).

Large software systems can have millions of source lines of code (SLOC). The Linux kernel 2.4.2 has $2.3 \times 10^6$ SLOC of C code alone [Wheeler02]. Windows 2000 purportedly has $29 \times 10^6$ SLOC [Lucovsky00]. A typical line of C code translates to about 6 abstract syntax tree nodes, so Windows 2000 is $0.18 \times 10^9$ nodes. A 1GHz CPU may provide 2 GIPs peak; a trivial analysis requiring 100 instructions per node then requires 10 seconds of CPU. If we somehow succeed in fitting AST nodes fit into single cache lines, the program still exceeds the size of the cache by orders of magnitude, so cache line accesses run at memory speeds, ensuring that simply touching the entire program costs 10 seconds. If we wish to give an engineer a fast turnaround tool after exhausting our algorithmic bags of tricks, our only available alternative is to provide parallel computation to alleviate these costs.

Actual computations are more expensive than these. We typically wish in match program fragments against patterns, apply multiple program transforms to a small fragment of source code to analyze/enhance it, or compute inferences over the source program. Such symbolic computations take hundreds of instructions per tree node when executed interpretively. Compiling such symbolic computations is clearly a good idea, when one has the time and energy to build such special purpose compilers. However, even such compilation can only alleviate the costs; scale ultimately can always demand more computational horsepower than we have available.

Ultimately, symbolic computation on large artifacts is simply expensive. It makes sense to optimize an engineer’s time by applying multiple processors in parallel to tasks of interest to the engineer. It is well known that it is extremely difficult to parallelize a program after it is implemented; the conclusions is that any large parallel symbol computation should be designed to run in parallel from the beginning. This is turn requires that the symbolic computation be coded in a system which makes it practical and efficient to encode parallel symbolic computations.

A happy consequence of the microprocessor revolution is that the incremental cost of additional microprocessors in a computer system is small. A typical CPU in a $1000 system costs retail $125 (AMD). The CPU vendors have reached market saturation; most computer system users already have one or more systems, and little motivation to buy another except as replacements. To maximize their profits, CPU vendors will, we think, increasingly turn to placing multiple CPUs in single systems with shared buses (“Symmetric Multiprocessing”) to leverage such system replacements. We can already see this effect in the marketplace; commodity 2 and 4 way SMP systems are widely available commercially. Initial attractions of such offerings are faster processing of graphics for games and pictures (Apple Power Mac G4), so this can be justified from a pure consumer focus. Other vendors provide larger-scale microprocessor systems as servers; Unisys offers up to 32 Pentium 4s in a single SMP system (E7000).

## 3 DMS

Early experience with Draco [Neighbors84], a multi-domain program transformation system, lead to automated porting between LISP dialects between two very different platforms [ABFP86]. The ideas behind the port eventually lead to a theory of Design Maintenance by capture, revision, and replay of transformational designs [Baxter90, Baxter92, BaxPidg97], driven by “designs” that explain program functionality, performance, implementation, and rationale [Baxter01b].

The vision of Design Maintenance as a practical, scalable tool is the driver behind the present commercial implementation of the DMS Software Reengineering Toolkit [DMS2001]. DMS is presently available for industrial strength reengineering and code-generation applications. Future versions should be capable of reverse-engineering low-level applications back to specifications [BaxMeh97].

The DMS Software Reengineering Toolkit is generalized compiler technology used to carry out practical, custom automated analyses, enhancement and code generation for large-scale software systems. The core component of DMS is a rewriting engine, enabling the principal benefit: reuse of generative knowledge cast as source-to-source transformations. However, the core issue for realizing a practical system is *scale*, along a number of axes. DMS provides for scale in encoding and working with multiple domain languages, at multiple levels of abstraction, for million-line source/target systems, using parallel computation as a foundation.

We believe that DMS provides the "right" generative technology base, in which reusable *implementation knowledge* (rather than code) is cast in the form of:

1) Specification parsers using robust GLR/Tomita [Tomita86, vandenBrand98] parsers, to simplify the problem of defining arbitrary human-readable specification languages and acquiring specification instances to drive generative processes. (The DMS vision includes the notion of graphical domains, but this is not presently implemented).

2) Source-to-source transformations ("component implementation knowledge"). These

    a. implement optimizations with a domain and refinements between domains [Neighbors84]. (Unlike Draco, procedural and mixed transformations are also practical under DMS.)

    b. are tightly integrated with the parsing and lexing technology to enable use with real languages (such as C++, Java, Progress)

    c. are executed by an associative-commutative rewrite engine, enabling good symbolic simplification for arbitrary arithmetic and Boolean formulas

3) High-level reusable knowledge about sequencing of transformations,

4) Domain-language specific analysis procedures usually implemented generatively by DMS from attribute evaluator specs

5) A *parallel* execution foundation (PARLANSE) to maximize the amount of computational horsepower available for analysis or transformation (the attribute evaluators are compiled by DMS into parallel PARLANSE code based on attribute information flow).

Some other generative methods, while widely available and therefore initially attractive, fail to handle scalability issues and will ultimately lead their users to long-term problems.

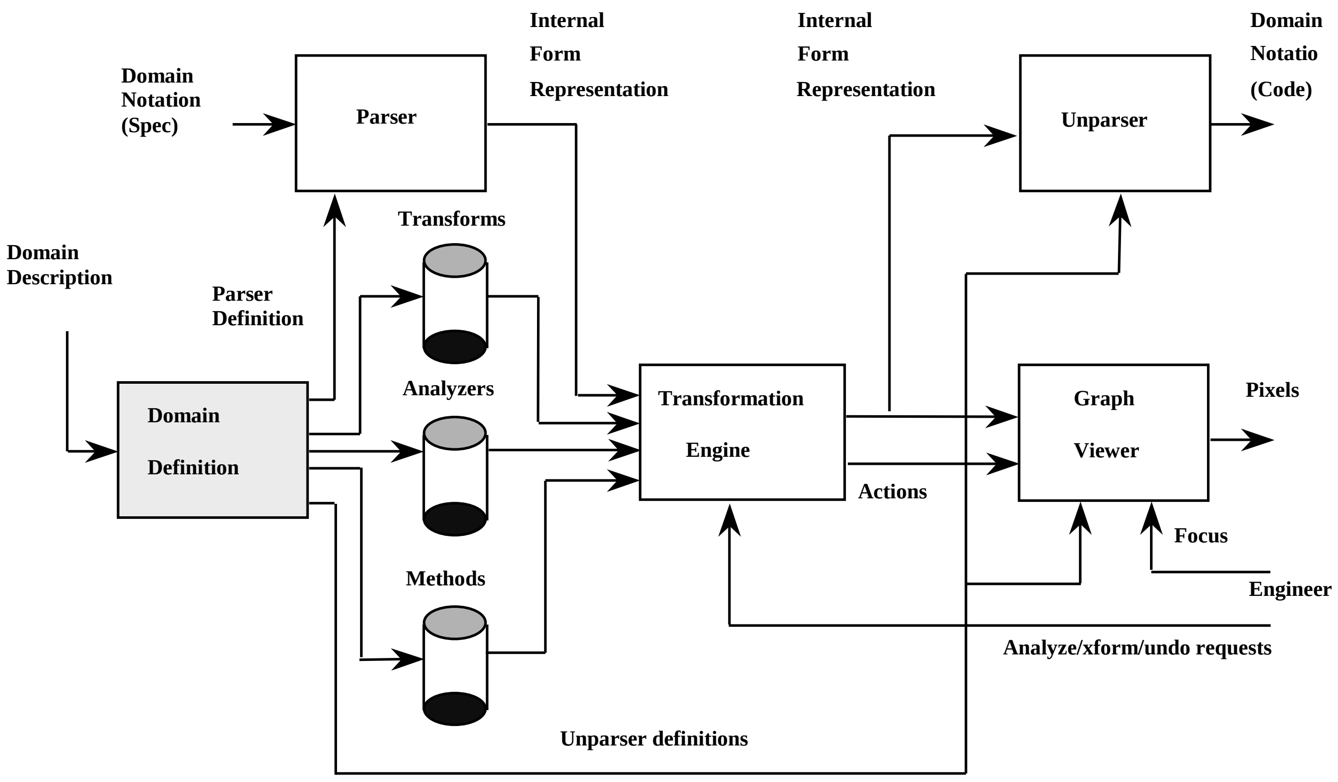

***Figure 1: DMS Architecture***

## 4 Applications of DMS

DMS has been used for a number of practical industrial applications, most focused on reengineering but some on generation:

1) Source formatting ("beautification"), and its converse, obfuscation by consistent identifier mangling

2) Removal of dead preprocessor conditionals from C source code (tens of rules) [BaxMeh01]

3) Detection of duplicate ("cloned") code (procedural analysis implemented in parallel), and macroized removal of clones (tens of rules) [BaxterEtAl98]

4) Insertion of test-coverage probes for Java, C and COBOL (with nearly trivial generalization to other languages) (roughly a hundred rules) [Baxter01a]

5) Porting JOVIAL (a legacy DOD language) to C, including translation of macros (commercial work in progress, several thousand rules). A 400K SLOC application has been ported and is under evaluation at the time of writing.

6) Generation of factory controller code from factory manufacturing process descriptions (commercial work in production; thousands of rules, several layers of domain languages and refinements, very strong Boolean formula simplification)

7) Generation of fast XML parsers from XML DTDs (work in progress; tens of rules)

8) DMS Self application:

    a. Generation of fast UNICODE lexers

    b. Generation of *million-line* parallel partial-order attribute evaluators, via DMS-based attribute evaluators

    c. Source prettyprinting

    d. Test probe insertion

Research work using DMS is focusing on reengineering Web sites [Ricca02].

Since DMS is neutral about the languages it processes, it is expected to be very useful in handling hardware design languages such as Verilog, and mixed language hardware-and-software co-designs. We have done some experimental work with Verilog and VHDL.

## 5 Irregular parallelism and Symbolic computation

Parallelism generally brings one of two models to mind: data parallelism and distributed computation. Data parallel models are those in which there exist large bodies of data which can be partitioned in parts that need essentially homogeneous processing. It is traditionally seen in physical model computations in the form of large arrays (e.g, a billion elements), which are partitioned into geometrically regular subregions and distributed across a number of processors, each of which processes its elements (typically applying only a few machine instructions per element, such as multiply-and-accumulate) and exchanges subregion edge data as needed with its assigned subregion-edge neighbors. These processors tend to synchronize at the end of each conceptual processing step (such as a matrix multiply).

```
(|; first  (<< second fourth) (+= x)
    second    (= z (fib x))
    third  (sort (. y))
    fourth (>> third) (= f y:x) )|;
```

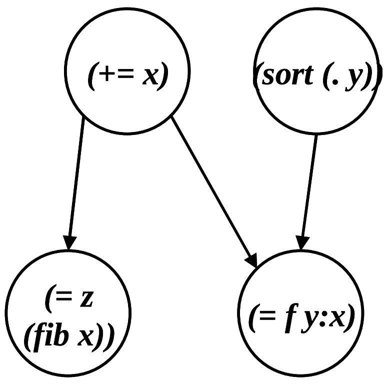

Figure 2: Partial order parallelism

Distributed computation tends to be seen in applications where the work can be statically divided into possibly heterogeneous parts and distributed across possibly heterogenous processors (some of which are designed to carry out their part of the computation efficiently, such as an FFT, or have access to a key resource, such as a database), which communicate as needed to manage the entire computation. Both of these classes are successful when the computational fragments are relatively large compared to the communication and synchronization costs.

There is a third class of parallelism which seems not widely discussed, which we call "irregular parallelism" (sometimes called "control parallelism). This class occurs when there is a large amount of computation to do, but the individual parts are heterogeneous, small to modest in size, are unpredictable in advance, and require considerable synchronization. Additionally, one may speculatively compute results (possibly in parallel), leading to the need for stopping useless speculative computations.

Symbolic computation for program analysis and manipulation seems to fit this model well. Here, primitive data elements are fragments of a program representation such as an abstract syntax tree node and its near children. Scale occurs because individual element computations can be expensive (pattern match, rewrite, attribute extraction/composition/movement), because rewriting for (e.g. Boolean) optimization can be NP complete, and because the program to be manipulated can itself be quite large (million-line C programs, or factory-controller code containing 25K complex Boolean expressions). Speculation can occur when searching a space of possible answers.

Considerable research in irregular parallelism (without speculation) occurred under the rubric of “dataflow computation” [Arvind93], and was the initial inspiration for PARLANSE. Dataflow research focused essentially on hardware supporting distributed computations with granularity roughly matching the primitives of the computation (add, index, compare, etc.). This did not lead to any commercially successful systems because of the requirement for special hardware, and the overhead (communication and synchronization) costs dominated the actual computation costs, and consequently the resulting computation rate was not very good. A key insight from this research was the notion of controlling the size of the computational grain to ensure that the overhead is only a modest fraction of the actual computational work done, to achieve efficient execution.

## 6 PARLANSE

PARLANSE (a pun for “language”) was designed to efficiently express irregular parallelism, and enable the construction of a large software systems (Thus, the name, PARallel LANguage for Symbolic Expression). This meant trying to achieve several goals:

1) Ensuring that the language could be compiled to very good code in the *absence* of parallelism
2) Simplify the specification of parallelism and its management
3) Integrate exception handling with parallelism
4) Controlling grain size to aid efficient computation
5) Providing software engineering support.

The first issue is a “do no harm” issue. Not every block of code can be highly parallel, nor can one achieve a meaningful speedup if one executes, in parallel, code which is necessarily significantly slower than that producible by good compilers for sequential languages. This suggests that most code sequences, especially the sequential portions, should be competitive in performance to that produced by good compilers for standard languages. As a consequence, we chose a “C” like foundation for PARLANSE, because much of the focus of the compiler community is on compiling C-like languages. Thus the

```
(define merge_sort_in_place (procedure [data (reference (array SomeType 1 dynamic))])
  (action
    (local
       (||
          (define small_fast_sort (procedure [lower integer] [upper integer]) ... )define
          (define merges_subsequences (procedure [lower integer]
                                                 [midpoint integer]
                                                 [upper integer]) ... )define
          (define sort_subsequence (procedure [lower integer] [upper integer])procedure
             (action (;; (ifthen (>= lower upper) (return) ); sorted subsequence
                         (ifthen (< (- upper lower) 20)
                            (;; (small_fast_sort lower upper) (return) );; )ifthen
                         (local (= [midpoint integer] (// (+ lower upper))// )=
                            (;; (|| (sort_subsequence lower (-- midpoint))sort
                                    (sort_subsequence midpoint upper)sort
                                 )||
                                (merge_subsequences lower midpoint upper)
                            );;
                         )local
                     );;
             )action
          )define
       )||
       (sort_subsequence 1 (upperbound data 1)) ; top level call on subsorting
    )local
  )action
)define
```

*Figure 3: Parallel Sorting by Divide and Conquer*

language has the following constructs, which were often modeled initially after those in C:

- Scalar data types: `boolean`, `integer`, `character`, `float` with the usual typed arithmetic over such values. A `symbol` type provides valueless but comparable identifiers which are globally unique (rather like enumeration value names).
- Compound data types: static `array`s, `structure`s, and tagged `union`s. Strings are simply arrays of characters. Unlike C, arrays always have a known size, and there are dynamic arrays which can be `resize`d at any time.
- Reference (pointers) to any type. Unlike C, no pointer arithmetic is allowed, athough one can take the address of a structure slot or static array element.
- Standard side effects such as assignment and incrementation of variables.
- Standard control constructs such as `ifthen`, `while`, `do`, function/procedure call, etc.
- Functions (and procedures). Unlike C, these take an explicit structure as an argument; one can build an argument list independently of calling a function. Unlike C, function signatures are explicit types and can be defined by a source library and used by name; this allows us to change signatures of libraries easily. Unlike C, functions are first class, lexically scoped and can be passed as values. Since programmers should not know much about sizes of values PARLANSE can pass arbitrarily big values to and from functions, and the compiler manages this efficiently based on the size of the result. PARLANSE programmers do know that it is cheaper to pass some values by reference. Because PARLANSE is a parallel language, activation records are heap allocated.
- Data access paths. Rather than complex syntax for array and structure access, PARLANSE simply simply separates access elements by a colon, e.g., `A:B:C`. The meaning is determined by context.

The second issue is to make parallelism easily accomplished by the engineer, and easily extracted from the application, so that the compiler doesn't have to struggle to locate it. We chose to do this by providing two facilities: a) the notion that expressions are dataflow-parallelism by definition, b) the programmer can easily specify standard parallelism idioms, and actually has to specify sequentiality in an effort to discourage sequentially-by-accident.

Exception handling has proven its worth repeatedly in building large complex systems (like DMS). It seems especially important to make exception handling work properly even in the presence to parallel constructs, and no practical language available when we started in 1996 (or even today to our knowledge) provides reliable exception handling that crosses parallelism boundaries (e.g., Java exceptions work only within threads). In retrospect, implementing exceptions in PARLANSE was one of our best decisions. DMS allows parameterized exceptions to be declared, raised, caught, inspected, and propagated. An extremely nice property is that exception handler overhead is exactly zero if no exceptions occur, so PARLANSE

```
(define parallel_visit (procedure [tree AST:Node]
                                  [visitor (action (procedure [node AST:Node]))]
   (action
      (local
         (|| [tree_walk team]
             (define visit_root_and_children (procedure [subtree AST:Node])
                (;; (do [i natural] 1 (AST:Nchildren subtree) 1
                        (draft tree_walk
                               visit_root_and_children
                               (AST:NthChild subtree I)
                        )draft
                       )do
                      (visitor subtree)
                   );;
             )define
         )||
         (;; (visit_root_and_children tree)
             (wait (event tree_walk)) ; wait for all grains to finish
         );;
      )local
    )action
)define
           …   (parallel_visit my_tree (action (procedure [node AST:Node])
                                         (rewrite node rules)
                                  )action )parallel_visit
```

*Figure 4: Parallel Tree Walk and Parallel Rewriting*

applications often use exceptions instead of explicit comparions to efficiently handle rare cases.

Simply being able to specify parallelism does not help if the grain size is too small (causing too much context switching time overhead) or too large (not enough parallelism to win). However, only the compiler is likely to have a really good idea of what the proper grain size is on an architecture. As a consequence, parallelism in PARLANSE is often advisory ("potential parallelism") rather than mandatory, and the compiler is allowed to coalesce parallel computations as it sees fit. So while PARLANSE has computation grains as explicit entities, often parallelism is not specified for grains, but rather for code, and the compiler handles all the grain management in such cases. The principal means by which this is achieved are the "potential parallel" construct and its generalization, the "partial order" construct. The current compiler does not coalesce grains, but heavy use of the potential parallelism construct leaves the next compiler free to do as it sees fit. What we have done is to arrange for the compiler to manage much of the implicit grain initialization and context switching. This enables the current PARLANSE compiler to create, schedule, run, stop, and destroy a grain with some 30 machine instructions. Consequently a block of 60 machine instructions in a grain can allow a 2 processor system to outperform a single processor system.

Good software engineering support can make building a large system easier and more manageable. Besides exception handling, PARLANSE offers modules, name space management, and good debugging facilities. Modules allow sets of named PARLANSE entities to be grouped into conceptual packages, with public and private definitions. PARLANSE provides nested name spaces with lexical access even for passed function values.

By providing assertion statements (`trust`)that are automatically converted into runtime tests by a compile-time debug switch, we made it easy for programmers to state checkable constraints. In addition, the compiler

```
(define parse_file_set (procedure [files (array (reference string) 1 dynamic)])
  (action
    (local (|| [live_parsers team]
               (= [parsed_files HashTable] ; atomically updateable hash table
                  (InitializeHashTable (action (procedure reference HashTableEntry))
                                            (= ?:parsed ~f) ; initialize slot to false
                                     )action
                  )InitializeHashTable )=
               (define parse_file (procedure [file (reference string)])
                  (local (= [needs_parsing boolean] false)
                     (;; (Update parsed_files file
                            (action (procedure (reference HashTableEntry)
                                    (ifthen (~ ?:parsed)
                                       (|| (= ?:parsed ~t) (= needs_parsing ~t)  )||
                                    )ifthen
                            )action )Update
                         (ifthen needs_parsing
                            (local [= [tree Node](Parse file))
                               (AST:ScanNodes tree
                                  (procedure [node Node])
                                    (ifthen (IsIncludeFile node)
                                       (draft parse_file (IncludeFileName node))
                                    )ifthen
                                  )procedure
                               )AST
                            )local
                         )ifthen
                     );;
                  )local
               )define
           )||
       (;; (do [i integer] 1 (upperbound files 1) 1
               (draft live_parsers parse_file files:i) ; put these files into work list
            )do
            (wait (event live_parsers)); wait for all parsers done
       );;
    )local
  )action
)define
```

*Figure 5: Parsing Many Source Files in Parallel*

inserts array range checks, invalid pointer checks (but not dangling pointer checks), and verifies that union tags are set properly when union accesses are made. Well-written PARLANSE codes tend to have such assertions every 10 lines or so, and violations of the assertions produce source location information directly. We find most bugs by reported assertion violations. We plan to add `pre` and `post` conditions to function signatures in the future.

Dynamic storage is a necessary curse in symbolic programs. You must have it, and you must clean up afterwards. We chose not to have a garbage collector, because in 1996 nobody knew how to build a parallel one and we could not risk the DMS project fate on an un-implementable idea (to date, there are still very few practical parallel garbage collectors, and they only exist in research systems). Instead, we provide the usual primitive facilities of `new` and `free`, adding the notion of a referenceable storage `pool` from which a `new` can be performed, and which can be freed as a unit. A `pool` is generally created when a subcomputation starts, and released when finished, so "leaks" of storage with respect to the pool are only transient effects. We believe this idea will still be useful, and certainly does not stand in the way of an eventual garbage collector. For many list-construction tasks where the list is modest size, our low-overhead dynamic arrays have turned out to be spectacularly useful and efficient.

PARLANSE has some other unusual properties. It distinguishes between values and entities. Values can be copied, processed, and compared for at least equality, and for which one cannot construct a reference. Entities are individually unique, to which only entity-type-specific operators can apply, always including "construct a reference". A classic entity found in most languages is the `variable`, which has operations to fetch and store its value. Other entity types represent items managed by PARLANSE such as `pool`s, individual parallel `grain`s of execution, and `semaphore`s. The `new` operator can dynamically create new entities as needed.

The language is entirely based on UNICODE, allowing UNICODE strings and names. In symbolic computation for arbitrary domains, it is convenient to name things they way the domain names them; if the language naming convention insists on a fixed character set, then some domains will have names that are difficult to express. PARLANSE allows identifier names to be arbitrary strings by use of judicious escape characters. Thus grammar token names, no matter how spelled, can be easily written in PARLANSE.

The oddest quirk is probably the LISP-like notation we chose to use for PARLANSE. All language forms have the syntax

(*keyword form1 form2 … formN*)*keyword*

The idea was to avoid the usual endless language-design

```
nested_class_declaration = nested_class_modifiers class_header class_body ;
<<BuildSymbolTableFrame>>:
     {
       IF nested_class_declaration[0].entity_kinds == MemberClassInInterface THEN
          nested_class_declaration[0].modifiers =
            AddOneModifierToModifiers(AddOneModifierToModifiers(nested_class_modifiers[1]
                     .modifiers,ModifierPublic),ModifierStatic);
       ELSE
          nested_class_declaration[0].modifiers = nested_class_modifiers[1].modifiers;
       ENDIF;
       class_body[1].symbol_space =
             CreateChildSymbolSpaceOf(nested_class_declaration[0].symbol_space,+1);
       AddToNodeToSymbolSpaceMap(class_body[1].,class_body[1].symbol_space);
       class_body[1].constructors = CreateSymbolSpace();
       class_body[1].wrapper_declaration =
         AddClassSymbolToSymbolSpace(nested_class_declaration[0].symbol_space,
                                 class_header[1].simple_name,
                                 MemberClass,
                                 class_header[1].node,
                                 nested_class_declaration[0].modifiers,
                                 VoidType,
                                 VoidSignature,
                                 class_body[1].constructors,
                                 class_body[1].symbol_space,
                                 nested_class_declaration[0].wrapper_declaration);
       class_body[1].label_space = nested_class_declaration[0].label_space;
       AddToNameResolutionTable(class_header[1].node,class_body[1].wrapper_declaration,
                                false);
     }
```

*Figure 6: Attribute Grammar Evaluator Specification to Process Java Class Header*

arguments about what punctuation to put where. In retrospect, we traded it for an argument about what keywords to put where; fortunately, the supply of keywords can be vast and so we never ran out of “punctuation”. The trailing keyword is optional and is called “coloring”; if present, the compiler insists it match the opening keyword, thus catching most nesting errors easily. Well-formed PARLANSE programs are like rainbows with matching colors on each end.

## 6.1 Primitive parallelism: `grain`

A parallel language must have some unit of dynamic execution, which in PARLANSE is a `grain`. Operations on grains include:

- `(spawn` *function data*`)` spawn creates a new grain executing *function* on the constructed function argument *data*., and returns a reference to a new grain.
- `(wait (event` *grain*`))` The invoker waits until the *grain* completes execution.
- `(abort` *grain*`)` The *grain* receives an asynchronous abort exception, forcing it to clean up and finish.
- `(doom` *grain*`)` The designated *grain* is told to destroy itself after completion; no further attention to it is needed.

PARLANSE offers the classic `semaphore` with `lock` and `unlock` operators which we do not discuss further. More interesting are `event`s, which can be `signal`ed to indicate the occurrence of an significant action, and `wait`ed upon for such a signal. Once signaled, further waits cause no delays. The operation `(event` *grain*`)` is simply a means to obtain a reference to the event associated with a grain’s completion. Events having values can participate in expressions as operands, and provide a direct means to implement certain dataflow computations.

Given grain operations (including `new` and `free)` and just `semaphore`s, one can program, with enough effort, arbitrary dynamic parallel systems. This is essentially the services provided by an OS threads package, albeit more efficiently because of careful code generation by the compiler. It still has the disadvantage of relatively high overhead over other PARLANSE mechanisms.

All the operations on parallel entities happen atomically, so there are entire classes of errors one cannot make in PARLANSE that could occur with a conventional threads package (e.g., this thread is aborted halfway through the process of aborting another thread itself). We have no further space to discuss this here, but PARLANSE has considerably more machinery for managing this.

The PARLANSE compiler and runtime system cooperate to ensure that no more than an adequate supply of grains exist at any one moment, to avoid the problem of having unbounded parallelism consume unbounded space. Built in time-slicing also ensures that arbitrary parallel execution succeeds even on single processor machines.

## 6.2 Parallelism with purpose: `team`

A `team` entity represents a set of grains (or sub`teams`) forming a parallel computation with an implicit purpose. Its value is in its use to manage the execution and/or termination of the entire parallel computation associated with that purpose. A `new team` starts with an empty set of member grains.

- `(draft` *team function data*`)` A new grain is `spawn`ed executing *function* on *data*. The grain is added to the team, and then `doom`ed..
- `(wait (event` *team*`))` The invoker waits until all grains in the team have completed.
- `(abort` *team*`)` Each grain in the *team* receives an asynchronous abort exception.

Teams provide a more structured approach to managing sets of grains than provided by standard thread packages. In particular, teams enable a manager computation to wait for a team result without knowing how many team members exist, or how/when those team members came to be (including being started by other team members!). It also enables speculative computations to be clearly grouped and easily stopped if later deemed inappropriate.

## 6.3 Statically scheduled teams

Dynamically constructed and managed parallelism is general, but has relatively high overhead. It is often convenient to harness statically-determinable parallelism. The PARLANSE compiler can manage this more efficiently precisely because it knows more, enabling it to issue instruction sequences to efficiently initialize, enque, and destroy the grains representing the static team.

Sequential code is the limiting case of a conceptual set of grains occurring in a serial order, and is coded as:

`(;;` *code1 code2 codeN* `);;`

for arbitrary blocks of code.

*Potential parallelism* declares that the set of computations can be executed in any order (including serially) and the result is acceptable. The compiler can ignore this parallel construct if the computations are too small. It is coded as

`(||` *code1 code2 codeN* `)||`

*Partial order parallelism* shows necessary ordering relations between computations to achieve a desired effect. Each computation gets a label and an optional ordering relationship in time (<< before or >> after) with respect to other labels in the partial order.

```
(;| label1l (<< labels) code1
    label2  code2
    labelN (>> labels) codeN  );|
```

Figure 2 shows the smallest partial order that cannot be coded with pure potential parallelism and sequentiality.

Partial order parallelism is also potential parallelism, because the computations can be ordered linearly and executed that way with no parallelism. It generalizes both sequential and potential parallism trivially.

## 7 Symbolic Parallel Applications

In this section, we briefly exhibit a number of parallel computations implemented in PARLANSE. All of these examples have run on 1 to 8 way SMP Wintel systems.

### 7.1 Parallel sorting

A parallel merge sort is shown in Figure 3. It uses potential parallelism to recursively divide and conquer the problem, by sorting both halves of a list, and then merging the sorted parts. Note the lexical scoping allows the recursive procedure to access the data array implicitly.

### 7.2 Parallel Tree Walk and Rewriting

Following the divide and conquer theme, Figure 4 shows a parallel tree walk, applying a `visitor` function bottom upwards. In the bottom of the figure, we invoke the tree walk, passing an anonymous procedure (`action`) as the visitor function that calls a rewriting engine with a set of rewrite rules, to apply those rules to the action's tree node argument. This implements bottom-up parallel rewriting.

### 7.2 Parallel Parsing

DMS is asked to parse software systems with thousands of files, often by giving it some source file subset which requires it to also recursively parse the *include* files mentioned by those sources. The code in Figure 5 accomplishes this. Since the number of source files is unknown, we employ a `team` to carry out the collective work. Each spawned subparser parses its designated source file, then inspects its tree for *include* nodes, and spawns children parsers as needed. The hashtable tracks which files have been requested for parsing, to avoid repeated parses. While not shown, the hashtable module is access and update safe for parallel threads.

### 7.3 Parallel Attribute Grammar Evaluation

Analysis of tree-structured artifacts can often be conveniently implemented using attribute grammars. DMS offers an attribute grammar evaluator to the analysis engineer. A useful analysis is the construction of a symbol table. Figure 6 shows a single attribute grammar rule for processing a Java class header. A DMS attribute rule passes (computed) values up and down the tree, and may also cause side-effects, such as updating the symbol table with the class name and type. The DMS Attribute Grammar Evaluator tool does a data flow analysis of the attribute rule computation, automatically determines a partial order that will effect the attribute rule, and synthesized PARLANSE code shown in Figure 7, with 9 partial order steps and up to 5 simultaneous units of parallelism

Our COBOL NameResolver is some 15000 lines of attribute grammar. The DMS attribute rule evaluator produces some 800,000 lines of partial order PARLANSE code to implement it. We believe this to be one of the largest parallel programs on the planet. It is in daily use.

### 7.5 Other Parallel applications

PARLANSE has been applied to a number of other parallel applications, which we can only mention here for space reasons:

- Clone Detection on 1.5 million lines systems
- Java Name and Type Resolution (above the attribute grammar level), which pipelines symbol table construction.
- Fast interactive determination of selected graphical object on complex displays by recursive space partitioning.
- Parallel rendering of software engineering diagrams with several thousand nodes [Quigley02].

## 8. Summary

This paper has discussed how symbolic computation on scale needs parallelism. An overview of an industrial strength, large-scale transformation system, DMS, was provided as a motivating example. The DMS implementation language, PARLANSE, is a parallel programming language that provides facilities for irregular parallelism as found in symbolic applications. We exhibited a number of running parallel programs used for various symbolic computations inside DMS.

We have preliminary numbers showing that indeed we are achieving some of the desired parallelism. While we have not achieved maximal parallelism under most circumstances, we think this is more a tuning problem than a fundamental technology problem. Being able to construct a reliable system as complex as DMS with parallelism is a fundamental step towards actually harnessing it.

## References

[ABFP86] G. Arango, I. Baxter, C. Pidgeon, P. Freeman, *TMM: Software Maintenance by Transformation*, IEEE Software 3(3), May 1986, pp. 27-39

[Arvind93] R. Nikhil, G. Papadopoulos, Arvind **T: A Multithreaded Massively Parallel Architecture*. International Symposium on Computer Architecture 1992, ACM Press

[Baxter90] I. Baxter, **Transformational Maintenance by Reuse of Design Histories** Ph.D. Thesis, Information and Computer Science Department, University of California at Irvine, Nov. 1990, TR 90-36.

[Baxter92] I. Baxter. 1992. *Design Maintenance Systems,* Comm. of the ACM 35(4), 1992, ACM.

[Baxter01a] I. Baxter, *Branch Coverage For Arbitrary Languages Made Easy: Transformation Systems to the Rescue!*, IWAPATV2/ICSE2001, http://www.semdesigns.com/Company/ Publications/TestCoverage.pdf

[Baxter01b] I. Baxter, *Breaking the Software Development Roadblock: Continuous Software Enhancement By Design Maintenance*, Software Design and Productivity Workshop, Vanderbilt University 2001, http://www.isis.vanderbilt.edu/sdp/Papers/ Ira%20Baxter%20(Breaking%20the%20Software %20Development).doc

[BaxMeh97] I. Baxter and M. Mehlich, "Reverse Engineering is Reverse Forward Engineering". 4th Working Conference on Reverse Engineering, IEEE, 1997.

[BaxMeh01] I. Baxter, M. Mehlich, *Preprocessor Conditional Removal by Simple Partial Evaluation,* AST01/WCRE2001, 2001, IEEE.

[BaxPidg97] I. Baxter and C. Pidgeon. *Software Change Through Design Maintenance.* International Conference on Software Maintenance, 1997, IEEE

[BaxterEtAl98] I. Baxter, et. al *Clone Detection Using Abstract Syntax Trees*, International Conference on Software Maintenance, 1998, IEEE.

[DMS2001] www.semdesigns.com/Products/DMS/DMSToolkit.html.

[Lucovsky00], M. Lucovsky, *From NT OS/2 to Windows 2000 and Beyond. A Software-Engineering Odyssey*, Keynote, 4th Usenix Windows Systems Symposium 2000, IEEE

[Neighbors84] J. Neighbors. *The Draco Approach to Constructing Software from Components.* IEEE Transactions on Software Engineering 10(5):564-574, 1984.

[Quigley02] A. Quigley, **Large Scale Relational Information Visualization, Clustering and Abstraction** Ph.D. Thesis, Computer Science Department, University of Newcastle, Australia, March 2002

[Ricca02] F. Ricca, P. Tonella, and I. Baxter, *Web Application Transformations based on Rewrite Rules*, submitted, Information and Software Technology, September 2002, Elsevier]

[Tomita86] M. Tomita, **Efficient Parsing for Natural Languages**, Kluwer Academic Publishers, 1988.

[Wheeler01], D. Wheeler, *More than a Gigabuck: Estimating GNU/Linux's Size*, Sixth International Workshop on Program Comprehension, 1998, IEEE

[vandenBrand98], M van den Brand, et al, *Current Parsing Techniques in Software Renovation Considered Harmful*, Sixth International Workshop on Program Comprehension, 1998, IEEE

```
(;|
  po1
    (ifthenelse
      (==
         _n0_:contents:value:attributes:entity_kinds
         EntityKinds:MemberClassInInterface)
      (local
         (= [_n1_ _Node_] (_create_attributes_for_existing_child_ _n0_ 2))
         (;;
           (;;
             (_evaluation_procedures_:(_get_evaluation_procedure_index_for_node_ _n1_) _n1_)
             (=
               _n0_:contents:value:attributes:modifiers
               (Modifiers:add_one_modifier_to_modifiers
                  (Modifiers:add_one_modifier_to_modifiers
                    _n1_:contents:value:attributes:modifiers
                    Modifiers:Public)
                  Modifiers:Static))
           );;
         );;
      )local
      (local
         (= [_n1_ _Node_] (_create_attributes_for_existing_child_ _n0_ 2))
         (;;
           (;;
             (_evaluation_procedures_:(_get_evaluation_procedure_index_for_node_ _n1_) _n1_)
             (=
               _n0_:contents:value:attributes:modifiers
               _n1_:contents:value:attributes:modifiers)
           );;
         );;
      )local
    )ifthenelse
  po3
    (=
      _n3_:contents:value:attributes:symbol_space
      (CreateChildSymbolSpaceOf
         _n0_:contents:value:attributes:symbol_space
         +1))
  po8
    (=
      _n3_:contents:value:attributes:constructors
      (CreateSymbolSpace))
  po13
    (_evaluation_procedures_:(_get_evaluation_procedure_index_for_node_ _n2_) _n2_)
  po10 (>> po1 po3 po8 po13)
    (=
      _n3_:contents:value:attributes:wrapper_declaration
      (AddClassSymbolToSymbolSpace
         _n0_:contents:value:attributes:symbol_space
         _n2_:contents:value:attributes:simple_name
         EntityKinds:MemberClass
         _n2_:contents:value:attributes:node
         _n0_:contents:value:attributes:modifiers
         Types:VoidType
         Types:VoidSignature
         _n3_:contents:value:attributes:constructors
         _n3_:contents:value:attributes:symbol_space
         _n0_:contents:value:attributes:wrapper_declaration))
  po17
    (=
      _n3_:contents:value:attributes:label_space
      _n0_:contents:value:attributes:label_space)
  po5 (>> po10 po17)
    (_evaluation_procedures_:(_get_evaluation_procedure_index_for_node_ _n3_) _n3_)
  po7 (>> po3)
    (AddToNodeToSymbolSpaceMap
      _n3_:contents:key:node
      _n3_:contents:value:attributes:symbol_space)
  po20 (>> po10)
    (AddToNameResolutionTable
      _n2_:contents:value:attributes:node
      _n3_:contents:value:attributes:wrapper_declaration
      ~f)
);|
```

*Figure 7: PARLANSE code generated from Attribute Rule*