# A New Framework for Software Development

D.B. Skillicorn[1]
Department of Computing and Information Science
Queen's University, Kingston, Canada
skill@cs.queensu.ca

September 1999
External Technical Report
ISSN-0836-0227-
1999-432

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared September 7, 1999


**Abstract**

Conventional software development does not allow proper design choices to be made when abstract machine interfaces are involved. The best design at one level of abstraction does not necessarily lead to the best design at the next, and subsequent levels. This problem has not been obtrusive in imperative, sequential programming, but becomes much more significant for targets such as superscalar processors, FPGAs, and parallel computers.

We suggest a perspective for software development where the goal of each phase is to produce all potential implementations to be passed to the next phase. We explore the implications of this simple idea, expressing the process of software design as a sequence of traversals of transformation spaces between two abstract machines.

[1]This work was supported by the Natural Sciences and Engineering Research Council of Canada

# A New Framework for Software Development

D.B. Skillicorn
Department of Computing and Information Science
Queen's University, Kingston, Canada
skill@cs.queensu.ca

## 1 Introduction

Software designers rarely write programs in the language of the processor on which these programs will eventually execute. Instead, a program is written in a programming language, which is in fact the instruction set of an abstract machine designed to hide the complexity and variability of the underlying system and hardware. Thus abstract machines, or equivalently programming models, correspond to programming languages. The task of software design is to translate from one language to another, and hence from one abstract machine to another.

For imperative sequential programming, the abstract machines are so well-behaved that we often forget that they are there. For instance, the processor that executes an imperative program may, in reality, contain a pipeline, may issue instructions out of order, and may move data between multiple levels of memory, but the abstraction of a processor executing a single, in-order stream of instructions with constant memory access time is a serviceable one. The usual cost model counts instructions executed, assuming falsely that floating point instructions take the same time as fixed point instructions and that memory access takes constant time, but the model is sufficiently accurate for most purposes. In fact, in costing programs we usually go further and ignore the constants entirely, concentrating the calculation of program cost only on loops and recursions.

There have always been some programs for which the standard abstract machine and cost model are inadequate. For example, designing a program for a memory-bound problem is difficult because the standard cost model does not capture the real costs of memory access. A programmer may choose the wrong algorithm because the memory behavior of the real machine is not reflected in the abstract machine.

Such inadequacies of the abstract machine are becoming increasingly important as processors themselves become more sophisticated. Whenever a processor is clever enough to improve performance other than by across-the-board speedup, some types of programs will start to behave in ways that the abstract model does not predict. Since programmers are using this abstract model to make design choices, this leads to performance surprises for individual programs and, on a larger scale, to poor design.

Building abstract models that accurately reflect the properties of parallel computers has always been difficult. This is partly because of the extra complexity of modelling a space dimension, and partly because of the non-linear behavior of communication (because of, for example, congestion). There are some abstract machines that are reasonably good models for wide classes of such computers, for example, MPI [2], BSP [5], and Cilk [1].

This paper addresses one of the major deficiences inherent in the current approach to software development. Each entity involved in designing a program tries to find the best design for the abstract machine below it, and then passes this design on to subsequent stages. The problem is that the best design at one level is not necessarily the best starting

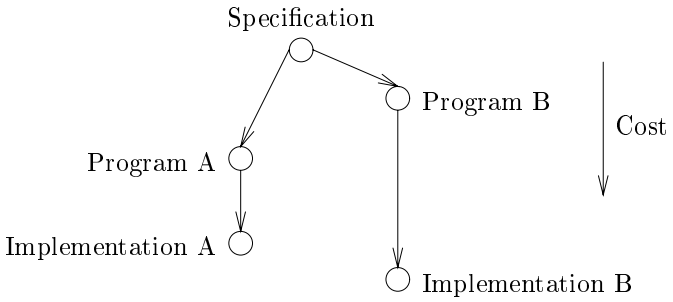


Figure 1: A Central Problem: the programmer's best design is not the best design overall

point for design at the next level. We suggest a new view of the development process. This new view is useful as a mental tool for thinking about the detailed processes involved, but may also make a difference to the practical tools used for software design. This work is motivated especially by the difficulties of parallel software design. One side-effect is the suggestion of new kinds of abstract machines for parallel computers.

## 2 A Central Problem

Any computational device is wrapped in a set of layers, each corresponding to an abstract machine of greater abstraction. A program written for eventual execution on the device must be translated from a form written for the most abstract of these machines, through a number of phases, to a form that corresponds directly to the physical machine. Even for the simple case of an imperative sequential program, there are several phases.

The process begins with a specification must first be translated from a specification language into a high-level programming language by a software developer. The program must then be translated into "machine language" by a compiler. The machine language is usually not the actual language of the processor but the instruction set interface of a family of processors. This machine language program is translated at runtime into a set of instructions whose ordering and other behavior is potentially quite different from the sequence of machine instructions produced by the compiler. In some settings, further layers may be involved: for example, a runtime system may add library code to the program, perhaps even dynamically.

The goal of the software designer is to produce the program whose eventual execution will be optimal. In other words, the program tries to drive the abstract machine corresponding to the programming language so that it will, in turn, drive the next lower abstract machine, and so on; so that the final, lowest-level abstract machine will execute as effectively as possible.

One of the problems that prevents good software design is the inability, in general, to optimize across abstract machine boundaries. Consider just the upper layers of the hierarchy of abstract machines. In Figure 1 a programmer has the choice between two algorithms, Algorithm A and Algorithm B. In the cost model of the programming language, Algorithm A appears to be the cheaper. Algorithm A is therefore implemented and passed

to a compiler. The compiler itself is translating from the abstract machine corresponding to the programming language to an abstract machine corresponding to the instruction set of the target processor. It may happen that the cost of translated Program B is less than that of translated Program A. The compiler is using a *different* cost model, that of the instruction set abstract machine, in which optimizations are possible that are not visible at the programming language interface. The programmer made the right choice, given the abstract machine visible to her; the compiler did the best job it could translating Program A; but together they have produced a sub-optimal program.

Of course, the problem could be avoided if the cost implications of the compiler's choices were visible in the programming language. But the compiler's choices depend on the cost structure that the runtime system will use to, say, schedule instruction sequences, so all of this information needs to be available in the programming language as well. It is conceivable that the hierarchy of abstract machines could be collapsed so that all of the details are visible to the programmer, but the complexity involved in programming is daunting. Many of the design choices in modern uniprocessors are made on pragmatic and *ad hoc* grounds, and so it would be extremely difficult to build abstractions that both accurately reflected these choices, and were in fact more abstract.

There is a second complicating factor – the cost of a program also depends on the sizes, shapes, and sometimes values of two sets of parameters, one for the program, and one for the target architecture. It will not always be the case that one algorithm is better than another for all of the possible parameter choices. For example, suppose that Algorithm A above has cost

$$(a_1 f(n) + b_1)i$$

where $a_1$ and $b_1$ are constants that arise from the 'structure' of the algorithm, $n$ is the size of the input, and $i$ is the time to execute an instruction on the target architecture. Further, suppose that Algorithm B has cost

$$(a_2 g(n) + b_2)i$$

The effect of the target processor speed on the cost of these two algorithms is relatively the same, so we routinely discount it in comparing the two costs. (Sadly, this is not the case with most architectural parameters.) Which algorithm is cheaper depends on the relative magnitudes of the $a_i$'s, $b_i$'s and $f$ and $g$. Usually, there will be some values of $n$ for which one algorithm will be better, and some for which it will be worse. In the absence of information about what input sizes will occur, the appropriate algorithm should be chosen at runtime, which means that both should be implemented. Each algorithm is optimal in some particular region. If we drew a graph with axes $n$, $i$, and cost, the target of our software development would be represented by the minimal surface in cost space, and the number of implementations required by the number of distinct minimal regions.

Developing optimal programs therefore requires building multiple implementations for two reasons: because different implementations will be best for different combinations of program and target parameters; and because our costs are not sufficiently accurate to be sure about which implementation is best for a given parameter set anyway. No wonder software development is hard!

Why has this problem not been more obvious in ordinary software design? In the standard imperative programming model, there is a fortunate separation between two kinds of transformations that can be applied to programs. The first kind change the order of

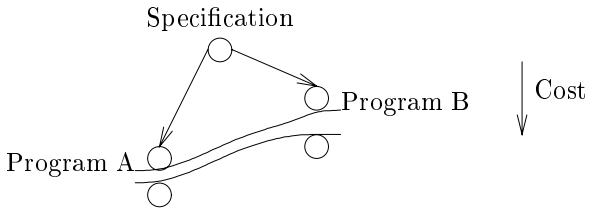


Figure 2: The Problem for Sequential Programming: the cost difference introduced by the compiler is at most a constant factor.

complexity of the program; the second change only the constants; and these two kinds of changes can be separated sequentially. In the cost expressions above, one algorithm must dominate for sufficiently large $n$, so we use the order convention and ignore optimality for small values of $n$. This is useful for developing complexity theory, but its practical usefulness depends on whether the input size for the application falls outside the "small" range or not. Fortunately, for sequential programming, cost expressions are usually simple, and functions such as $\log n$, $n$, and $n^2$ have magnitudes that differ markedly even for small $n$. Software developers make transformations of the first kind; compilers make transformations of the second kind. The problem setting becomes as shown in Figure 2. It is still possible for a programmer to choose the "wrong" design, but the suboptimality can only be by a constant factor, and we have agreed in the complexity theory to ignore such factors. The design choice can be poor, but only for executions using "small" data.

This assumes that the underlying processor really is a von Neumann machine, which was never quite true, and is increasingly false. The problem is encountered as follows: the compiler has a standard strategy for changing programs in ways that it thinks are optimizations. When these optimizations do not actually improve program performance, programmers are forced to write their programs so that the compiler will be unable to discover these optimizations, and so will translate the way the programmer wanted it to. Since it is not always easy to tell what sequence of transformations a compiler will apply and why, this interaction becomes an infuriating one for the program designer.

We have presented this problem as it occurs at the top level of the design hierarchy, but the same problem repeats at all levels. As we have seen, the programmer cannot tell which program to write to cause the compiler to generate the best program for the runtime system; the compiler cannot tell which program to generate to cause the runtime to generate the best instruction sequence (for example, to hide memory latency); and the runtime cannot tell which instruction sequence will be best for the issue logic to keep the functional units busy.

Two trends are increasing the impact of the problem. First, targets are increasingly variable, both in properties and over time. The boundary between hardware and software is now quite mobile; an abstract machine that would earlier have been implemented by hardware may now be a software emulation or an FPGA, and both are far easier to change. A major motivation for having abstract machines is to insulate the higher-level abstract machines that software and compilers target from changes in the lower-level machines. We do not want to have to rebuild the entire sequence of abstract machines every time the

lowest-level platform changes.

Second, parallel computing is becoming increasingly important and, as we have already mentioned, there are few good abstract machines that capture their properties in a way that can be exploited for software design. At present, parallel programs must almost always be tuned carefully after they are built to achieve reasonable performance.

These trends mean that far more abstract machines deviate from the von Neumann model for which an acceptable design methodology exists. For new abstract machines, the problem of optimizing designs across boundaries is important. The two trends come together in metacomputing systems, where the goal is to write programs that will execute on widely-separated, heterogeneous machines whose availability and performance vary continually. Building abstractions that allow software to be designed for these environments is challenging indeed.

The consequences of being unable to design across abstract machine interfaces are: lack of clarity about algorithmic choices, lack of clarity about goals and hence strategy of translation from one abstract machine to another, and the wasted work that occurs when an odd design has to be chosen at one level so that the next level will choose a better design. All of these significantly increase the costs of developing software, and decrease the effective performance of the software when it is developed.

# 3 Traditional Software Construction

Traditional software construction is regarded as a sequence of phases, the first carried out by a software designer, and the remainder by software tools. In terms of abstract machines, each phase is a translation from a program written for one abstract machine (the upper abstract machine) to a program written for another abstract machine (the lower abstract machine). For example, in sequential programming, a programmer translates a specification (an extremely abstract kind of program) into a program written in a high-level language. A compiler translates this program into a machine language program. A runtime system finds an execution sequence for this program. Issue logic finds the best pattern of despatches for this instruction sequence using a small window of lookahead.

Of course, this process is not usually entirely sequential. A program may be revised after it has been compiled, or perhaps even after it has been executed. So there is an iterative component as well.

What is the goal of each of these entities? Each one tries to find the translated program which is best (i.e. cheapest) in the cost model of the lower abstract machine. To do this, it has access to some kind of transformation or refinement system that defines how programs may be altered to preserve their meaning while changing their cost. This process may be formal (a refinement calculus [3], compiler optimization rules) or informal.

Each entity is constructing a tree, whose root is the upper abstract-machine program. A branch corresponds to the application of a "rule" which generates a new textual form of program from the previous one; and each node in the tree corresponds to a new program.

Each phase is finished when the upper abstract machine program has been translated entirely into the language of the lower abstract machine, and the cost of the program is good enough. Conceptually, perhaps, the entity might search for the optimal program, but this almost never happens in practice. The "best" version found is then passed to the next phase.

The central problem posed in the previous section shows that this process cannot work well, because the goal of each phase is ill-formed. No entity can know which version of the program is best to pass to the next phase without knowing what the next phase's transformation and cost structures are like. This amounts to a complete breach of the abstraction. Such breaches have been proposed: for example, making compilers aware of details of processor design and memory hierarchy is an active topic for research [6].

Let us now examine what happens when programs are derived in more detail. For clarity, we will imagine that this process occurs in quite a formal way. Even when it happens more informally, the same mechanisms are at work.

There are two underlying pieces of machinery that are essential for program derivation. The first is a set of transformation rules which define when one textual program can be changed into another form that is semantically equivalent (transformation) or better (refinement). The second is a cost model which gives the cost of each textual form of program, usually in terms of execution time.

An initial program forms the root of a tree of those programs that can be derived from it using the allowed transformations. The cost model enables each of these programs to be assigned a cost. Indeed, the cost model and the transformation system can sometimes be integrated so that each transformation is labelled with its cost-reducing direction.

Even in a conventional setting, the process of transformation is not straightforward. For example, simply reducing cost is unlikely to find the optimal program. So the tree structure that results from transformation is not one where the lowest cost forms are at the leaves. The cheapest program may appear anywhere in the derivation tree.

So why does the development process have this fundamental problem? The reason is that cost models, for good reasons, do not give a completely accurate picture of the cost of different textual forms. The process of transformation is really a language translation problem – we begin with a program (or specification) written in the language of one abstract machine and we want to find a program written in the language of some more concrete abstract machine. Each of these languages has basic operations, and composition operators. Let us define the "upper" abstract machine as $A$ and the "lower" abstract machine as $B$. The transformation system may contain rules that allow $A$-programs to be changed into new $A$-programs, rules that allow $A$-programs to be changed into $B$-programs, and rules that allow $B$-programs to be changed into new $B$-programs. The goal of this stage of the transformation process is to replace an $A$-program with the best completely $B$-program.

The cost model gives the cost of each $B$ operation and the way in which costs behave under the composition operators of the $B$ language. If these costs and composition properties were completely accurate, then there would be no difficulty finding the best $B$-program. For example, if the costs of individual operations are quantitative in nature, and these quantities add across compositions, then $B$'s cost model will be well-behaved. This is why the ordinary von Neumann cost model is useful; costs are instruction counts. However, if costs depend on *arrangements* or if it is possible to cancel work at lower levels, then cost models behave more erratically. For example, cache behaviour is history-dependent (hence dependent on arrangement) and so programs that depend on the presence of the cache have costs that differ from that predicted by the simple von Neumann cost model.

Let $\diamond$ be a generic composition operator in the language of $B$, and $\bowtie$ a generic composition operator in the language of $C$ that implements $B$. Then rules with the effect

$$B_1 \diamond B_2 \rightarrow C_1 \bowtie C_2 \bowtie C_3 \bowtie C_4$$

where $B_1 \to C_1 \bowtie C_2$ and $B_2 \to C_3 \bowtie C_4$ will allow well-behaved cost models provided that $\bowtie$ is additive on costs.

There are four structural problems in the correspondence between abstract machines that will break accurate cost modelling. First, composition operators in $C$ may not be additive, a property that I have elsewhere called convexity. For example, in many parallel computers the cost of communication is highly nonlinear because of congestion. The parallel composition of two pieces of program may be made cheaper by making one of the pieces more expensive by reducing its communication rate, and hence reducing congestion. This makes it impossible to design each of the pieces by minimizing its (local) cost.

Second, there may be specialised implementations of certain compositions of $B$-operations, for example rules of the form

$$B_1 \diamond B_2 \to C_1$$

Here the cost of the composition is smaller than the apparent cost of its pieces. Such specialised cases can always be dealt with by defining a new $B$-operation equivalent to $B_1 \diamond B_2$ and costed appropriately. However, this mechanism is often cumbersome if the specialised version reflects some wider underlying facet of the $C$-machine. Note that such rules can exist in more complicated forms such as

$$B_1 \diamond B_2 \diamond B_3 \to C_1 \bowtie C_2$$

as well.

Third, the algebra of $C$-operations may sometimes be cancellative. For example, there may be translations of the form

$$B_1 \to C_1 \bowtie C_2$$

and

$$B_2 \to C_3 \bowtie C_4$$

but $C_2$ and $C_3$ cancel each other out, so that

$$B_1 \diamond B_2 \to C_1 \bowtie C_4$$

(This can be regarded as a special case of the previous pattern, except that it tends to be ubiquitous and hence impossible to treat using the special case mechanism.) This kind of behaviour in $C$ is typical of parallel computers where each operation is preceded and followed by data communication operations that place data in the memory of processors. For some compositions of computational operations, the result arrangement of one may be suitable as the input arrangement of the next, so that these data communication operations cancel out.

Fourth, some transformations may only be allowed or may have different costs depending on the context in which they occur. There may be rules of the form

$$B_1 \diamond B_2 \diamond B_3 \to B_1 \odot C_2 \odot B_3$$

where $B_2$ cannot normally be transformed to $C_2$.

All of these cases will make the cost model inaccurate and therefore make it impossible to transform an $A$ program to *the* optimal $B$ program.

Note that a well-behaved transformation system is one that can be represented as a transformational grammar that is linear-bounded in the *cost* (rather than the length) of terms.

# 4 A New Model for Software Construction

It is clear from the discussion above that a process in which each phase passes one best version of a program to the next phase is inadequate. This is particularly obvious if, at the early stages of software development, we want to design in a way that is portable. Because software lasts a long time, some of the target machines for a given piece of software may not yet have been invented; in this case it is clear that we cannot possibly know what "best" will mean. In this section, we propose instead to equate a program with *all* of its versions, and treat this object as the goal of software design.

We begin with a specification as the central object of interest. A specification describes a desired computation in some language, and therefore includes, at least implicitly, an abstract machine as its context. When we talk about a specification in the normal way, this abstract machine is often a Turing machine or something at that level of abstraction, and we simply leave it out of our thinking. A specification is one of three kinds:

1. it is infeasible, so that there is no abstract machine operation that will compute it (we know from the Halting Problem that there are many specifications we can express but not compute, so that there is a gap between the language and the abstract machine in this case);

2. a specification can be non-executable, that is it is well-defined but not a description of a computation of the implied abstract machine (for example, it might be a (precondition, postcondition) pair);

3. a specification can be executable, in which case we normally call it a program.

A transformation system provides the connection between the language of an upper abstract machine and the language of a lower abstract machine. In the space between two such machines, a specification is considered executable when it written solely in the language of the lower abstract machine. The goal of the translation process is to alter the specification, initially written in the language of the upper abstract machine, into the language of the lower. The "rules" of the transformation system relate one set of syntax to the other. Note that this need not be a one-way process – it is perfectly reasonable, and many transformation systems allow it, to transform a program back into a (non-executable) specification.

We redefine the concept of a specification to be equivalent to *all* of its implementations (relative to a given abstract machine). For an infeasible specification, this set will be empty. For a feasible specification, the set will be non-empty, possibly infinite. The distinction between a non-executable and executable specification disappears, since a non-executable but feasible specification nevertheless has one or more equivalent implementations. Another way to think of this is that a specification is considered equivalent to the entire tree of feasible programs that can be derived from it using the transformation system. In a natural way, we also redefine the concept of a program to be equivalent to the set of all implementations derivable from a given specification.

We now restate the transformation process as: given a (member of a) specification, derive the program that corresponds to it. The starting point for each phase of software design is a set of upper abstract-machine programs (a specification by our new definition); the goal is to produce the entire set of implementations of that specification in the language of the lower abstract machine (a program by our new definition).

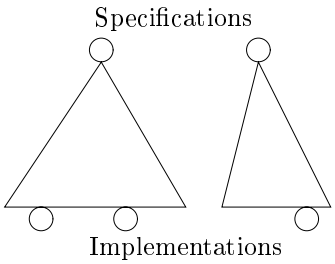


Figure 3: A Typical Transformation Space

In the traditional scheme, the goal was to search a tree of specifications reachable from the initial specification and find the best executable one. Now the goal is to find *all* of the executable nodes of the reachability tree and pass these on to the next phase. Note that the executable nodes are *not* the leaves of the tree, for transformation does not necessarily proceed in the direction of either implementability or cost reduction.

This change of viewpoint solves, or perhaps avoids, the problem of optimization across abstract machine interfaces. The possibility of handing over a suboptimal starting point to the next phase is eliminated, because all starting points are handed over (at least conceptually; we postpone practicalities until later sections). Clearly this process cannot continue for ever; the processor cannot, in the end, execute more than one implementation of the program. However, the point at which the single implementation is selected for actual execution is as late as possible, when it is finally clear which one that should be. Of course, the cost of doing this is that we must somehow keep all of these different implementations available and pass them around.

Since a phase begins with more than one textual form of specification, and computes a set of textual forms (implementations) as its result, the structure that must be explored by a designer is the set of trees, rooted at each element of the specification, and induced by the available transformation rules. Let us call this a *transformation space*.

# 5 Transformation Spaces

A transformation space is the set of paths from each upper abstract machine implementation (which we now regard as a specification) to a set of lower abstract machine implementations. Its structure is induced by the available refinement or transformation rules which allow new implementations to be built from old ones.

It may be possible to derive a particular (lower abstract machine) program from more than one specification, but there is no point in doing so. Thus we may eliminate some transformation paths from the transformation space. Its structure becomes a set of trees, each rooted by one element of the specification, as shown in Figure 3.

Cost plays a vital, but subtle, role in transformation spaces. Each abstract machine has some architectural parameters (instruction execution speed, memory latency, number of processors, and so on). The cost of an implementation on that abstract machine is an expression in those parameters, as well as some "program structure" parameters. From the

perspective of the current transformation space, these program structure parameters are properties of the initial specification, but in the previous phase they were the architectural parameters of the upper abstract machine.

Each element of the specification is labelled with a cost expression in terms of the parameters of the upper abstract machine, and also with a predicate in these parameters describing when the specification is valid (that is, a precondition). This predicate naturally arises because the reason for having more than one element of the specification (that is, more than one implementation for the upper abstract machine) is that each will be best over a given range of parameter values.

Each executable specification (that is, element of the program, or implementation on the lower abstract machine) has a cost in terms of both lower abstract machine ("architectural") parameters, and upper abstract machine ("program structure") parameters. This cost is an expression, not a value.

The transformation system for a transformation space may provide useful information about costs; for example, individual transformations may be annotated with the change in cost they induce, or perhaps only with their cost-reducing direction. However, this does not by itself help in traversing the transformation space because many good (i.e. cheap) implementations occur at the end of transformation sequences, some of whose steps are cost-increasing. See [4] for an extensive discussion of this issue.

It is entirely possible that an element of a specification is redundant in the sense that all of the implementations that can be derived from it can also be derived from other elements of the specification. We would like to be able to reflect this information at the next higher level, but this does not seem to be possible unless it has some effect on the cost structure. (This is unsurprising because there is an element of arbitrariness of deciding which of two paths from a specification element to an implementation to use.)

Sometimes it may be both natural and useful to combine derivations for more than one lower abstract machine in a single transformation space (for instance, a shared-memory and distributed-memory parallel program might share some common structure). This is a natural way to address portability, since a great deal many of the implementations may be programs for all of the lower abstract machines, and the work of creating them need only be done once. In this setting, some implementations are part of a program for one lower abstract machine, some are part of a program for the other, and some are part of both. This shows that there is in fact a continuum: at one extreme, all implementations are regarded as part of a single program; at the other, each implementation is regarded as a program for a particular lower abstract machine that is a specialization of the more-general abstract machine. This specialization is usually a predicate on architectural parameters (rather than more qualitative factors such as shared-memory) and is the origin of the predicates on elements of specifications at the next level down. This is illustrated in Figure 4.

With this perspective, the set of transformation spaces itself forms a tree whose nodes are transformation spaces, and whose edges are abstract machines.

The properties that make a transformation space attractive are exactly those you would expect: only a small number of transformations possible at each point; only a single way to do each thing; and simplicity.

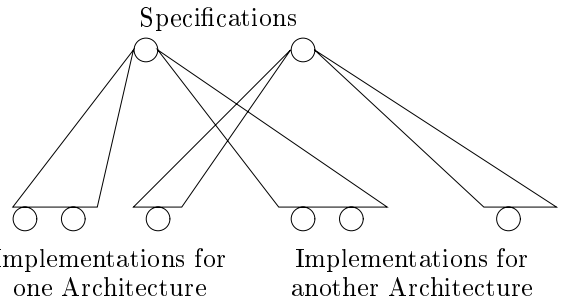


Figure 4: Transformation Space for more than one Abstract Machine

# 6 Using Well-Behaved Cost Expressions

When the interaction of cost measures and transformation systems are well-behaved, that is none of the problems described in Section 3 occur, the number of implementations required can be pruned. We have observed that the cost expressions that label each implementation are functions of upper abstract machine parameters, lower abstract machine parameters, and constraints on the upper abstract machine parameters. For the sake of illustration, let us take $n$ to be the upper abstract machine parameter, $p$ the lower abstract machine parameter, and suppose that, for a particular element of the specification, we have a constraint that $n > 1000$. (The specification may consist, say, of two elements, one constrained by $n > 1000$ and the other by $n \leq 1000$.)

Assume for the time being that the cost expressions are linear in $n$ and $p$. Consider cost space, with two axes indexed by $n$ and $p$, and a cost axis. Each implementation corresponds to a hyperplane in this space. There is a convex hull of minimal surfaces in this cost space, with each implementation that is cheapest for some values of $n$ and $p$ corresponding to a facet on this convex hull.

In fact, each hyperplane corresponds to a whole family of implementations with different values of $p$ say. However, all members of the family have the same cost expression. By Brent's theorem, we can always rearrange a parallel program with one value of $p$ to give another with a smaller value of $p$. In general, therefore, we only need one implementation on each facet if we can find the others by trivial transformations of this kind.

The number of facets of the minimal convex hull cannot exceed the number of implementations. In practice, we expect many potential implementations not to be part of the minimal convex hull because their costs exceed other implementations at all points in $n$-$p$ space. Further, the greater the restrictions on $n$ and $p$, the smaller this space is to begin with.

One approach to program design is therefore to construct this convex hull during the process of constructing and traversing transformation space, discarding implementations that are too expensive based on implementations found so far. This a a kind of branch-and-bound search, complicated by the fact that we cannot prune a branch whose current leaf is too expensive, because some of its further transforms may still be cheaper.

Of course, many cost expressions will not be linear in the parameters. The number of "facets" becomes correspondingly larger.

The implementation corresponding to each facet has a cost which is then passed down to the next phase. This cost is a combination of the cost expression and a constraint which is the union of the constraint inherited from its root specification and the constraint that defines the facet. An interesting, but unexplored, question is whether this constraint should also include the union of the applicability conditions associated with the transformations that were applied to generate this implementation.

It may happen that the performance of some abstract machines depends on properties that are not easily encoded as magnitudes. For example, if a program's behavior depends on the values, as well as the sizes, of its inputs; or if an architecture has discontinuous performance based on some switch setting, then the approach outlined in this section will not work.

In general, we expect that the process of finding good implementations is simplified if the number of lower abstract machine parameters is small. It is no surprise that many practical transformation space tools try to reduce the number of such parameters by assuming that cost does not depend on them. For example, the PRAM model assumes constant time memory access to memory from any number of processors, which clearly cannot be true in practice.

# 7 Choosing Abstract Machines

In conventional software design, we try to choose abstract machines at the "right" level of abstraction to try and avoid the problem of optimizing across them. Two strategies are used. The first is to choose the level of abstraction to get monotonicity in the architectural parameters. This makes it easier to be sure that if one program is better than another at one level, its implementations will be better than the implementations of the other at the lower level. As we have seen, imperative sequential programming relies on this kind of monotonicity. The second strategy is to reduce, in absolute terms, the number of abstract machines between the software developer and the hardware itself. This is the motivation for projects that expose all of the details of the processor and memory to the compiler, so that only one abstract machine boundary has to be crossed. It is also the motivation for functional programming, where (at least in principle) software developer and compiler see the same abstract machine and transformation rules, and the software developer can hand the text of the program off to the compiler at any point. In fact, the transformation space and strategy for traversing it are too weak in most functional languages for this to be really true, so programmers must find a good "starting point" from which the compiler can begin.

In our new model of the software design process, there is no penalty in introducing extra layers of abstraction, since the original problem of crossing the interfaces they create has disappeared. However, we would like abstract machines to be "thin" in the sense that they don't provide multiple ways of doing the same thing. This has the effect of reducing the number of "facets", which correspond to the number of implementations in a program.

The ability to add extra abstract machines makes this approach fit naturally with the skeleton approach to parallel computing, in which it is common to replaces the operation of a parallel computer with a single machine-wide skeleton that encapsulates a complex computation.

Notice also that this approach replaces data refinement by machine refinement. Machine refinement is inherently more attractive since it can be done once, and used by many

programs, whereas data refinement must be done for each program (until truly reusable libraries become commonplace).

# 8 Representing Programs

Since a program is now a set of implementations, some thought must be given to the concrete representation of these programs. There are three levels of difference that can be distinguished between any two implementations making up a program:

- The implementations describe different algorithms. It is hard to be precise about what constitutes a different algorithm, but it may be useful to separate two algorithms if insight is required to see that they are functionally equivalent. Keeping different textual representations would seem to be unavoidable here.
- The implementations describe different arrangements of computation (and memory access and communication in some settings). Here a single textual representation can stand for a possibly-exponential number of different arrangements. The single representation need not necessarily be resolved into the actual best arrangement in order to be passed to the next phase (a closure in a functional program is an example of such a representation being passed to another abstract machine). This makes the creation of some subsets of the implementations of a program lazy, potentially postponing the choice of arrangement until the choice of the single implementation to be executed. (Of course, an algorithm with exponential running time may be required to select the best arrangement from this program representation; however, this resolution only needs to be done once, rather than at each phase.)
- The implementations describe the same arrangement with different parameter values. As we have already discussed, we can simply regard these as a single implementation.

It may therefore be possible to keep a much smaller set of explicit representations of implementations than a simple view would suggest. It is also the case that a given computation is often textually smaller the more abstract the machine for which it is intended. The overheads of multiple representations may therefore not be excessive.

# 9 Building Programs in Pieces

We have talked of software design so far as if it were a monolithic activity. Any practical software design methodology must also permit programs to be constructed in pieces which can then be joined together to give a well-behaved single program. This is required so that large software can be built by different groups, and is also the key to software reuse, for there is no way to tell the difference, at assembly, between program pieces developed by a separate group today, or taken from a library.

The key to dividing specifications and assembling the resulting programs is the provision of composition operations. Such operations are reflected in the transformation system by the presence of rules that say how to break specifications into smaller ones (whose resulting programs will automatically fit back together properly). For example, in sequential imperative programming, the simplest composition operation is sequential composition (;).

The quality of the composition operations available has a strong effect on the usability of an abstract machine. Composition is well-behaved when the amount of program state involved is small, and the amount of mental state involved is also small. One of the weaknesses of message passing is that a single thread cannot be composed with another thread in a straightforward way, since all the sends and receives must be checked to make sure that they "match", which actually requires analyzing the data flow of the two threads. No wonder deadlocked parallel programs are easy to write. Higher-order functional programming involves compositions where there is significant mental state required, which may partly explain why imperative programmers often perceive functional programming as difficult.

A transformation rule that introduces a composition operation can often (but not trivially) be used to generate a rule that introduces residuals. Such laws answer questions about the difference between a specification and an existing implementation – what else is required to turn the existing implementation into one that satisfies the specification? Such residual laws allow library programs to be introduced into derivations.

Of course, in this new view, a library program is actually a set of implementations, with the same upper machine interface. What this view makes clear is that an upper interface is not sufficient. Each implementation in a library program also have a lower abstract machine interface, and it is these interfaces which are the basis for selecting which elements of the library program are actually used in the main derivation. In the conventional view, a library program is regarded as saying "I can do this", but what it is really saying is "I can do this on an abstract machine of this kind". The missing information is a kind of precondition, while the conventional interface as a postcondition.

# 10 Discussion

The primary goal of the new model presented here is to make it possible to really design software, that is to make choices between different designs using criteria that reflect reality. One requirement for this is cost models that accurately reflect the actual cost of operations. The imperative sequential model is open to criticism on the grounds that memory cannot really have constant time latency; the functional sequential model on the grounds that there are often much more efficient ways to represent data structures than those in the model; the PRAM model on the grounds that constant time access to shared memory is not possible in the real world; and many parallel programming models on the ground that they fail to account properly for congestion in communication.

However, the issue that we are addressing is a deeper one, which occurs even if cost modelling were perfect, the inability of a designer at one level of abstraction to see the best design possible at the next (and subsequent) layers. This problem is pervasive, although little noticed in conventional sequential software because design can be divided into two phases, one that alters orders of complexity, and one that only alters constants. Any approach that insists that a designer's job is to find the "best" program and then hand it on to the next phase is doomed to failure.

We propose an alternative in which, conceptually at least, the outcome of design at each stage is the set of all implementations that could be optimal for some range of values of the "program structure" parameters and the "architectural" parameters. This at least makes it clear what a designer is supposed to be doing; although it is not yet clear whether the idea can be made practical.

It is encouraging that some of these ideas are already present in embryo form in other contexts. Those who design software for non-traditional hardware and systems, especially exceptionally variable systems such as FPGAs, have already to be aware that today's best program may be mediocre tomorrow. In a similar way, software designers for large-scale heterogeneous systems must build programs that execute on a wide range of different computers, the choice of which may only take place after the program has begun executing. They must also be prepared for the available resources to change even in mid-execution. Tunable and adaptive programs are therefore special cases of programs as we have defined them.

# References


[1] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded languages. In *ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, June 1998.

[2] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming.* MIT Press, Cambridge MA, 1994.

[3] C. Morgan. *Programming from Specifications.* Prentice-Hall International, 2nd edition, 1994.

[4] D.B. Skillicorn. Architectures, costs, and transformations. In *Constructive Methods for Parallel Programming, CMPP'98 (associated with Mathematics of Program Construction*, pages 1–15, June 1998. Fakultät für Mathematik und Informatik, Universität Passau, Report MPI-9805.

[5] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[6] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, September 1997.