White paper

Transparent Analogy as a Foundation for Language

A small but revolutionary computing project

Copyright Lawrence J. Dickson, 6 April 2006. Circulated by permission.

Contents

1 Summary

Current computing paradigms can handle neither deep complexity nor asynchronous races. Computing designs do not enforce global correctness, nor do descriptions of components suffice to define their behavior. Things work only because of a “black art” of deeply embedded driver and kernel code, both critically important and concealed from the master design.

The dominant paradigms have a fundamental flaw, which makes revising and adding to them a fruitless endeavor. This flaw is metaphor, or the uncritical application of constructs to dissimilar realities. Because theory drives computing — a program is a long-drawn-out expression of precise theory, which is slavishly acted out by the computer — the failure of the constructs makes predictable design impossible.

The cure is to require analogical constructs, whose theory permits usages drawn only from those features that are similar among the realities they are programming. A list of specific metaphorical failures implies a set of language and design principles which must apply to all levels of computing, from power-up to power-down. The implied paradigm is not unknown, though it practically passed out of use in the 1990s, and was never applied to the full breadth of a system, but only to disconnected subsets.

The new paradigm is "resource-oriented", partially known in the past as "process-oriented", and it has a simple, flat, mathematical character. An instance of it, well known in the past, was CSP/occam on the Transputer ([10], [11]). My previous published work [5] proves the concept of extending it to full system breadth. The defining concepts are so clean that this is not a great task, as long as the work is done on a single exemplary system and set of peripherals, with care taken to ensure it can be extended analogically to others.

The project is the Transparent Analogical Complete Language Instance of the Resource-Oriented Paradigm, or TACLI for short.

2 Analogy and metaphor: the critique

For the purposes of this paper, "analogy" equates similar things, and "metaphor" equates dissimilar things. In metaphorically applied computing constructs, the points of dissimilarity must be tracked in addition to the standardized behavior of the construct. I will call the code that deals with these dissimilar features "critical code", while code that deals with features of the kinds shared by all the realities programmed by the constructs will be labeled "regular code".

Because metaphorical constructs allow usages inconsistent with the critical code of some of the realities programmed with those constructs, correct usage of the constructs can lead to incorrect program behavior. Therefore, in addition to design and language expertise, an arcane skill in critical code and related usages is demanded of the programmer. The constructs hide the critical code, and the usages can involve theory (code snippets) in widely separated and separately developed parts of the program.

Now let this code becomes more complex, with layers dependent upon one another, although their creation was widely separated in time or space. Two bad consequences follow. First, the most critical code, the least accessible to the construct descriptions and most dependent on usages, migrates to the deepest layers, as programmers fix ordering and usages and encapsulate the results in protective API specifications. This becomes kernel and driver code — most critical, least understood. Second, the arcane knowledge required for the correct working of each layer becomes less accessible to the programmers for all the other layers, and among related snippets of critical code, widely separated in the source, it becomes harder and harder to enforce correctness. This results in the instability of large projects. The difficulty of enforcing global correctness increases much faster than program size, which has caused the burgeoning of "worm's eye view" test-driven and plan-driven program management techniques ([1], [2]).

What is lost is the assurance, common in building architecture and car architecture, of being able to snap together matching components using only their specs to get a functioning whole. A whole category of software (middleware) expresses the impossibility of this dream in computing architecture.

Now consider a few specific examples of failing metaphor. I will present the construct (theory) as a metaphor of the dissimilar target (reality) in each case, and in each case the theory will be too permissive or incomplete to model the reality correctly.

2.1 Infinite as a metaphor of finite

There is a famous example in economics, the theory of “perfect competition” based on an infinite number of vendors, modelling the reality of three vendors (GM, Ford, and Chrysler). Thus,

$$\infty = 3$$

Computing is not far behind. All stack-based dynamic call languages like C, and all languages that offer dynamic memory allocation (malloc), permit an uncontrolled infinite theoretical construct that draws from a finite resource pool. Worse, most system implementations allow these to draw from a system-wide pool, and thus different instances of program run will get different results in general.

Underneath the metaphor is a hidden state machine, the stack pointer or the memory allocation table, which (if you are lucky) will return you an error if you attempt a correct usage that overflows the resource pool. This is an instance of “microcorrectness”: the error is a valid response, but offers no aid in implementing a provably correct global design. The hidden state machines, and the critical code that handles them, are a typical instance of kernel or driver code, buried deep beyond the control of ordinary programmers.

2.2 Sequential as a metaphor of parallel

Most programmers think the sequential execution stream is the most natural form for a program, but hardware designers know better. On the transistor level, combinatorial logic is far simpler to implement than clocked logic. But it does not stop there. Very few, if any, computing systems are truly sequential.

If you have peripherals, the firmware on the peripherals is executing in parallel with the code on your CPU. If you have interrupts, then multitasking (or virtual parallelism) is happening on the CPU itself, even if your operating system is supposedly single-tasking like DOS. Each interrupt service routine (ISR) is an event-driven program of its own, communicating with the main or user programs. And if you have DMA, FIFOs, serial buffers, or the like, then your computer contains resources that are under the control of multiple parallel code sequences.

The inadequate sequential paradigm makes do with “system calls” to hide the critical code handling the interface with other parallel execution streams. This dates back to the ancient Fortran PRINT statement. The simplest technique is just to wait, which can stall execution at these special commands. Later developments like buffering make for faster execution and more arcane code response, because the hidden state machines become more complicated and have more failure modes.

2.3 The call metaphor

Arithmetic calls (functions, procedures, methods) like sin(x), log(x), atan2(x,y) are analogues of one another. Accepting parameters in range, they give a result deterministically. But these calls are only metaphorically related to stateful system calls like open(x), close(x), read(x,y). You must call the last three in the proper order — open THEN read THEN close — which implies that a state machine, hidden out of sight, is necessary for their proper operation. Typically it will involve hidden parallelism of the 2.2 type, and different device types will behave very differently even though their calls look the same, adding yet another layer of metaphor.

For example, IO to the same disk or file system can be invoked by independent Linux processes, but an inadvertent attempt to drive a magnetic tape from two processes led to interesting results at a former employer’s lab.

Object-oriented (OO) methods are particularly susceptible to metaphor, because they almost always have composite action including IO and other hidden state, and subtle differences are encouraged by OO polymorphism. But the call metaphor extends far beyond OO, to C, Fortran, Basic, and almost all other procedural languages, and to all known operating systems including primitive ones like DOS. The simple encapsulation provided by the standard list of formals, applied uncritically and without discrimination, has exacted a heavy price in design uncertainty.

2.4 Template as a metaphor of real entity

The term “object” in the phrase “object-oriented” is itself a metaphor, since outside the computing world, “object” normally refers to a real, enduring thing with coherent features. I will call this a “real object” in the discussions that follow. The object of computing is a data-organizing template like a C structure, which I will call a “template object”. Manipulation of the data found in an area of computer memory organized according to this template is what a method does, at least while it remains fully analogical in the sense of 2.3.

Object-oriented computing is used to manipulate real objects, although its constructs and methodology are based on template objects. This confusion extends to other terminology like “components” and “interfaces”. As a result, not just state machines but actual physical devices like disks and keypads enter the realm of hidden, critical code.

One characteristic problem is that real objects endure across time, and their features exhibit an inertia that is subjected to coherent state changes. A method, by contrast, cuts across the momentary state of its template object memory, capturing one or two states. The family of methods misses the intrinsic organization that connects the states across time, especially time spent outside the control of the code of this family of methods.

A second characteristic problem is that different kinds of real objects cut across each other’s controls. For instance, a resource allocator shared by several kinds of devices must itself exhibit a coherence across time, but its code will be intermingled with that for all the device objects.

OO code, like the IO call code of 2.3 and the driver code of 2.1 and 2.2, can attack the first problem by enforcing microcorrectness — bracketing the method call with state checks or locks. This is seen in the “preconditions” and “postconditions” imposed by the “software contracts” methodology. Like “test-driven” project management [1], it shows loss of hope that global design correctness can be enforced. It is like Calvin’s dad’s method for testing a bridge design... drive larger and larger trucks across it until it breaks!

The “interfaces” of Java and the later “aspect-oriented” paradigm show an attempt to attack the second characteristic problem. As admitted in the Wikipedia article, the tendency of aspect-oriented code to modify critical code at widely separated locations means it leads to uncontrolled side effects. The process-oriented methodology proposed below, by enforcing the enduring and coherent analogue of a real object, eliminates this vexing problem with ease.

2.5 The architecture and design metaphors

As the previous subsection makes clear, metaphor extends to terminology, where computer jargon uses a common term (like object or component) in a technical sense that is inadequate to model the common term. This is widespread, and helps to give software engineering its bad reputation as the “sick man” of the engineering disciplines. The main value of this proposal will be to open a path to literal applicability of some of these vital terms, like architecture and design.

In physical engineering, a design, or architectural description, of a thing is sufficient to enforce correctness on anything complying with its specifications. If you place struts of the specified strength in the positions demanded by a bridge’s plan, and connect them as specified, the bridge will stand up. This is also true of computer hardware.

It is rarely true in software. A “naive” implementation of a program architecture or design specification will almost certainly result in an unreliable or failed product. Vast, arcane knowledge of the parts and their hidden relationships is needed to achieve robustness.

The metaphor problems in the previous four subsections are sufficient to explain software design's inability to enforce correctness. They actually imply something worse: software techniques, applied to standard paradigms, are unable to check for correctness in any but the simplest cases. The side effects of hidden critical code cannot be predicted in cases of complexity beyond the scope of one programmer, and often can't be predicted even in code written by one programmer over a large span of time.

This can spell doom for large projects that are required to produce reliable results, like the air traffic control software port (see [2]). New programs required to exchange data with a wide array of existing systems are famously prone to failure, like the recent Medicare prescription fiasco, and many software upgrades by governments or major corporations. Even drivers, which are made as standardized as programmers can manage, fail to behave predictably often enough that only multibillion dollar developments by companies like Microsoft can come close to smooth operation. And even there, occasional glitches are expected and accepted.

There was one exception to this rule. Transputer hardware and software, in its heyday, was famous for behaving according to design, and that was subject to a harder than usual test — for its native programming language, occam, made it easy to insert test points throughout an architecture without affecting device behavior. This family of products was also the one whose paradigm avoided the metaphor problems of the previous four subsections. This paper will go on to show that one follows from the other, and how the particular success can be made general.

3 Transparent analogical model of the resource-oriented paradigm

The key to good law is to codify customary usages. Similarly, the key to solving the metaphor problem is to revert to clean customary methodology that has been known since the dawn of software. At every point the analogical principle must be our guide; and within that constraint, the simpler the better.

Transforming the metaphors 2.1 through 2.4 into analogies is clearly required (a necessary condition). That it is also sufficient can be shown by the construction itself, and the fact that it works to eliminate metaphor 2.5, and make real design possible. We are aided here by the very wide experience of the occam/Transputer research community, massively active worldwide during the 1980s and 1990s, whose constructs and results are still on the record. That extension of this rigorous work to the complete system is possible has been demonstrated by my DOS work [6], which serves as a proof of concept.

The test of the proposed complete language specification is its simplicity, a test resoundingly passed by the legacy occam/Transputer work which it extends. The attempt to keep up with the ravages of metaphor always results in massively expanding language and OS specifications; examples are Java, Linux, and Windows. The successfully transparent and analogical language and OS specification, by contrast, will start small and remain small, because it is clearly correct and complete.

Analogy will confirm the value of our key constructs, for they will each be exemplified in a rich profusion of independent useful entities long known in the computing world, both hardware and software. In each case, the robustness of the analogies has already been proven by the work of the occam/Transputer community and my DOS work. In fact, getting it working in non-multitasking DOS (from DOS 3 to DOS 5) serves as proof of concept for both user systems and embedded systems, since DOS is the point of intersection of both.

3.1 The box

The attack on metaphor 2.4 begins with a concept familiar to every programmer: the program. The following text template bursts from the fingers of each C hacker without thought:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
  /* enter program here */
  exit(0);
}
```

And yet in the world of nested software paradigms, it has not been made clear what distinguishes a program from other code blocks, such as a function call, an OO method, or an event response.

My analogy for a program is a hardware component, a "black box" which boots up, communicates with the rest of the hardware in a well-defined fashion, and shuts down. This analogy is exemplified in the ".PGM file" language of the Transputer configurer, where the program-equivalent is a separately compilable process, which can either be run on a dedicated separate CPU with physical links and peripherals (hardware black box), or run as an occam process sharing a multiprocessing CPU with other processes (software program). My DOS work ([5], [6]) proved that a robust analogy between this and DOS programs can be enforced, using a very simple scripting mechanism (the "daughter" file) in the *absence* of multitasking oversight by the OS.

The overarching analogical construct is the "box" (for "black box process", but also referring to the resource and information box in Figure 1). A box is a process (a finite set of resources subjected to transformations by a finite set of commands over an extended but finite time), subject to the following conditions:

3.1a. It is "sequentially unified": its execution trace has a single program-controlled start command and a single program-controlled final command (which may differ dependent on branch values and entry points), and all its commands follow in sequence except that it is permitted that parallel execution streams may be opened under program control after the start, subject to the condition that each member of such an array of parallel execution streams be closed before its parent terminates and, after all are closed, the box's execution trace resume its sequential nature. (This is exemplified by the CSP/occam "PAR" construct and differs from forking and spawning in conventional OSs, which create uncontrolled child processes subject to metaphor 2.1.)

3.1b. It is "sequentially maximal" or "independent": the initial command of the box does not follow by program necessity from a sequentially previous command, nor is the final command of the box followed by program necessity by a sequentially subsequent command. This permits the PAR children of a box themselves to be boxes, since both sides of the enclosing construct are non-sequential. (The children initialize and terminate in non-deterministic order.) The programs called in sequential or nested order by a non-multitasking OS also qualify as boxes, as long as these calls are arbitrarily ordered through something like a command line or script, and not forced into relationships of dependency by a syntax or resource requirement.

Consider the bash script

```
if [ -e file1 ] ; then
  rm -f file1
else
  echo "file1 does not exist"
fi
```

The commands “if”, “then,” “else”, and “fi” would not qualify as boxes: they need an overarching script and other programs to run as designed. The individual “[ ]”, “rm”, and “echo” programs bracketed by these branches could each qualify, as could the entire script.

3.1c. It is “transparent”, with explicit boundary conditions and no side effects. Its initial resources and data are enumerated in full by such means as calling sequence (software), boot program and peripheral specification (hardware), or OS shared resource specification (program), and startup values are supplied by the enclosing box. Its final resources and data are similarly enumerated and returned to the enclosing box. Changes in resource ownership or data may be transmitted in or out across box boundaries only through a specific enumerated set of channels, under explicit program control both inside and outside. The channels are themselves resources owned by the enclosing box. No global variables accessible from outside the box may be shared inside except as explicit parameters.

This setup is shown in the diagram below.

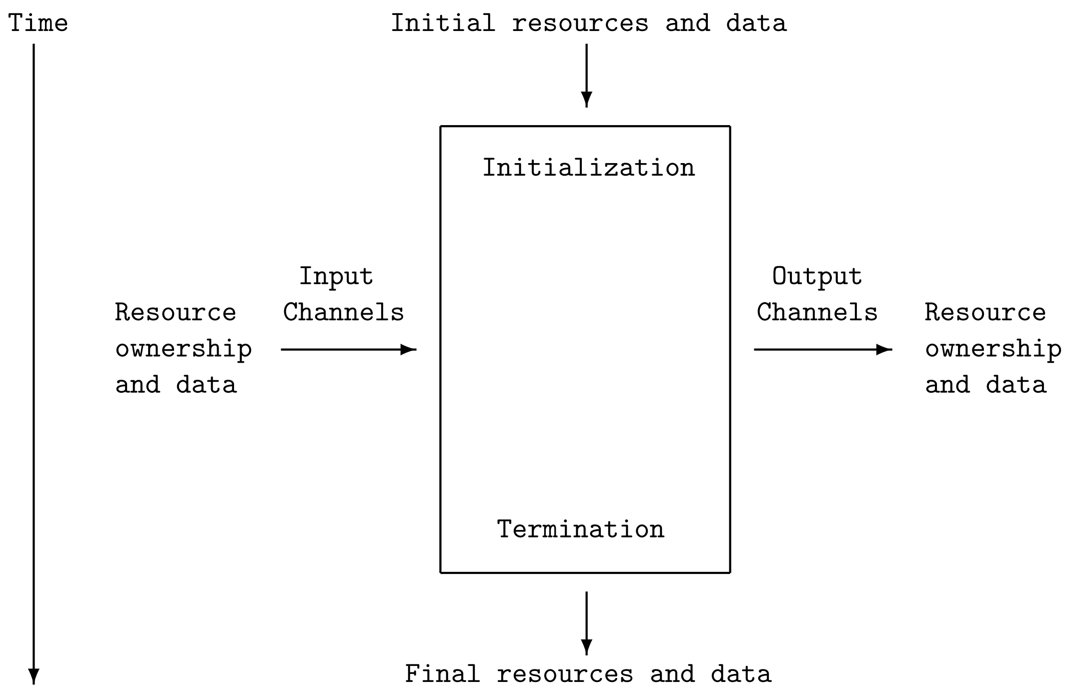


Figure 1. Box data and resource flow over time (strictly enforced)

Box creation is exemplified, and the concept thoroughly proved, in the usages of the “occonf” tool and the “.PGM file” syntax of the occam toolset provided in the late 1980s by Inmos for transputer arrays. See [7]. It is important to note that the process code accessed by the occonf tool (which is an occam compiler variant restricted to parallel constructs) must be linked from “separately compiled” object files, which enforces the transparency condition by preventing external references to global variables except via the formals list. This has to be so, in order to allow the configurations to span arrays of distributed CPUs whose only shared resources are a set of unbuffered point-to-point links and (sometimes) a shared reset/analyze/error signal triplet.

MASM-type DOS assembly programs, written with their natural CODE, DATA, and STACK segments, are static in nature and approximate a box structure ([8]). This cleanness is lost in higher-level compiler output, not because it is higher-level, but because it is metaphorical. TACLI compiler output will regain it, and in fact where the tame/wild concept is enforced (see 3.3 below), it will be possible to share one stack.

Box creation on the operating system level is exemplified, and the concept proved, by the work leading to [5]. Using no more than the ancient CPM/DOS program segment prefix (PSP), without OS multitasking capability, standard DOS programs are run both sequentially and in parallel as boxes communicating in a way fully analogical with both the CSP/occam processes and with independent PCs using network connections. A simple set of calling parameter conventions applied to "daughter" programs enumerates both shared system resources (stack usage at 30 bytes per program) and communication channels.

3.2 The channel

Unlike "box", the name "channel" is pretty standard. The common meaning of a communications channel is a good analogue of the strict TACLI and CSP/occam meaning, which includes the requirements that the channel be point-to-point and unbuffered. The occam channel is already analogical, being used both for software process communication and for physical hardware links, two or (late T9000) four wires each. A channel is unidirectional, while links in Transputer hardware are bidirectional full duplex (two channels each).

One clear expansion by analogy is to apply channel to any truly multiprocessing hardware connection. In this case, in Transputer terminology, it is called a "link". For instance, in DOS a printer with printer interrupt is modeled precisely by a unidirectional channel. There is no buffering, and each byte receives an interrupt acknowledge. The timeout-free patience of standard printers, holding the queue all weekend until someone refills the paper bin, is typical of channel behavior.

Further channels can be modeled over a serial connection and, faster yet, over a DMA-mediated bus connection with transmission-done interrupt. The fact that the communication is unbuffered does not require an interrupt per byte, as long as the DMA machinery can be told how many bytes to expect in the block transmitted. All these channels are implemented on the interrupt level in my DOS code. With a very few hard interrupt response code templates, all kinds of peripheral connections can be modeled by links.

The soft channel is implemented by a memory copy and process rescheduling. This forms a good analogy with software sockets and pipes — good but not perfect, and where there is disagreement the channel is better. There is no operating system involvement in the channel. A channel, when relinquished by a terminating box, remains in existence to be picked up by a later box if desired, while the box at the other end of the channel waits. This improves on the centralized and destructive character of Unix pipes, which "blow up" the process at the other end. Also, the unbuffered character of the channel allows precise control of communicating process response. Buffering can be introduced by explicit programming if desired.

Another analogue of the channel is the Unix file descriptor in general. The ubiquity of the file descriptor in Unix corresponds to the ubiquity of the channel in TACLI. See 3.6 below for details.

An expansion on the CSP/occam model is TACLI's capability of passing resource ownership, as well as data, on the channel. This permits the controlled use of pointers (data block resources) and the zero-copy handling of large amounts of data, similar to block device drivers in standard OSs. Resource ownership must, in every case, be controlled by a carefully designed state machine; hence the name "resource-oriented paradigm". In resource-passing boxes, termination must include the restoration of borrowed resources in a provably correct way. Locally static resource ownership (see 3.3 below) means this can be done by a variation of occam "abbreviation" that is block-atomic and capable of invalid blocks. One simple version of this is analogous to a remote procedure call (RPC) across the channel.

A cursory examination of Transputer ".PGM" files shows that the channels modeling hardware connections are defined on the outermost levels of the nested code. This means that former critical code behavior, including interrupts, is moved to the outermost and most accessible test points. This is the opposite of the metaphor 2.2 and 2.3 behavior, in which critical code is buried deep in the call nesting, practically inaccessible to testing, due to uncontrollable influence of globals in the enclosing calls.

In Transputer occam, the entire transmission through a data channel can be captured by an extra soft process in the ".PGM" (configuration) file, leaving all the deep code unchanged, and hardly affecting performance. Compare this with standard drivers: I challenge anyone to train a disk block device driver to send voluminous diagnostics out an ethernet link without heavily affecting disk performance.

The requirement that a channel be owned by an enclosing box outside the box or boxes that use it is sufficient to organize all communication, even hardware communication that becomes possible as soon as power is turned on. The outermost (boot) box that executes code is enclosed by a further box that executes no nontrivial code of its own but serves as a mounting for hardware resources. This turns the whole ".PGM" file into a box, whose only contents are declarations, definitions, and a PAR construct. It can also be applied to the behavior of a host machine which activates peripherals at power-up, or subsidiary CPUs as on a B008 with TRAMs.

3.3 The resource road map

The infinite metaphor 2.1 is completely eliminated by occam, which uses a little-known algorithm that allows calls to be made without a stack. The price is a finite compile-time limit that is imposed upon all parallelism (and, in occam, recursion is modeled by parallelism) and upon all calling sequences, call depth, and memory allocation. In fact, all resource allocations are static in occam. This not only allows each library object module to give an exact byte count of the memory it will need. It even permits memory usage to be assigned statically from the bottom up, so that innermost calls get lowest addresses — which allows a very simple form of caching in the Transputer: fast low addresses.

TACLI relaxes the CSP/occam requirement a little to permit "locally static" behavior. This means that at any time, all boxes running in parallel, even if currently descheduled, "know" exactly which resources are assigned to them, and do not step outside them. Finite limits may be imposed at load time (of the box to be initialized), not necessarily compile time; this allows an OS command line. Resource ownership may be passed by channels, as mentioned above in 3.2.

TACLI also allows limited recursion (or forking or spawning) within a fixed overall limit. By allowing only one "heritage" (current innermost box) at any given time to have these "wild" properties, a command line or GUI or other user control point can be supported without any resource fragmentation. The wild heritage can also run 2.1-metaphorical programs (like C programs) if desired. This is further discussed in [5].

Because the "tame" (non-wild) heritages cannot apply the infinite metaphors, TACLI compilers must use the occam constructs in generating nested or library routines. They will thus have empty relative stack at all points internal to calls, even deep within the nesting. This makes the application of descheduling, caused by channels or otherwise, trivial and OS-free (though transitory locking is required). It also means that the memory and resources shared by the box-globals at the topmost declaration level of a box behave in much the same way as a template object, while deeper-nested declarations use a fixed pool of scratch memory.

Stack will still be used, but in a lean and predictable fashion. When DOS PSPs are used for program complexes, each program gets a fixed 30 bytes on the stack. High-priority (hard interrupt service) code on the Transputer gets 6 words, and similar allocations can be made for OS-based interrupt code. If there are multiple priorities, each infinitely higher than those below it, several such stack assignments can be made. Between the interrupts and the tame heritages' stack allocations, the wild heritage is allowed to take up as much stack as it wants, subject to the limits imposed by metaphor 2.1.

3.4 Bootup and configuration

Booting fits the box and resource model. The booting box is enclosed by a resource-only box that declares and defines all resources that survive power down. Booting itself, instead of being hidden and mysterious, is expanded into a well-defined and transparent program that activates resources in proper order.

The three common types of boot may all be dealt with in this manner. ROM boot treats the full operating system (and perhaps the whole usable programming) of the system as a power-down resource, so that the only active part of the boot code is a wait for ready state. Network boot is the opposite, with minimal power-down resource definition, leading to transmission of all active code through a data channel that starts in a special, code-loading state. BIOS boot is an intermediate case, with the power-down resource including a stripped-down set of primitive operations that is capable of accessing a cascade of boot code supplied by the user on data storage media. There are combinations of these three models, such as the BIOS-based "ROM Basic" found on old IBM PCs.

All of them can be explicated by the nested box model with resource declarations. The resulting program description may well be massive in typical BIOS cases, and the number of nestings may be surprisingly large, with things like a 512-byte boot sector getting more attention than hitherto. The key is to trace all resource activation.

As an example, consider network booting as exemplified by the Transputer boot from link, code found in IOCTOOLS\EXAMPLES\BOOTSTRP in the 1990 toolset. Booting is sequential. Omitting channels, declarations and definitions, it takes the following C-like pseudocode form:

```
{
  void bootloader() {
    network_loader();
    user_program();
  }
  bootstrap();
  bootloader();
}
```

In fact, this pseudocode is inadequate: the bootloader code (by G. S. Panesar 11/14/1989) is by structure a tiny operating system that could run a sequence of non-nested programs. Its structure is:

```
{
  initialize;
  LOOP:
    load program code through link;
    run program;
    set up for next program;
    goto LOOP;
}
```

The links are a general resource whose protocols in the load program code, the network_loader run, and the user_program run, are all distinct. The network_loader passes code to nodes that are farther away than its own node, and then as its final act, sets up for code intended for its node. As coded the user_program is not quite a box, since its specifications are dependent on final network_loader input, but that input could equivalently be moved up into the bootloader set up code for strict compliance.

When combined with resets, the bootstrap/bootloader, overwriting itself, may be considered as an unchanging resident OS given the assumption (whose alternative is insane behavior) that control of the resets and peeks comes from a single, sequential source. That allows BOOTLINK and MEMSTART to keep a constant value. This is important for the general analogical analysis of error response and post-mortem debugging.

Solid analogy holds here with ethernet-based network loading on diskless workstations, and also with primitive COM-port code loading in such devices as the HP DOS Palmtops. In those cases, BIOS is also involved.

Configuration is treated as a separate step in OS-based systems, but it is in the TACLI model an outer program run, spawning "tame" children. DOS for instance offers the following pseudocode ([8]), where TOPLEVEL is TRUE if an instance of command_com is the top one (loaded at boot) and FALSE if it is, or is nested within, another command_com's user_program.

```
void command_com() {
  noexit = TRUE;
  while (noexit || TOPLEVEL) {
    command_prompt();
    if (command == exit) noexit = FALSE;
    else user_program();
  }
}
boot_sector();
sysinit();
command_com();
```

The DOS driver loading takes place under control of CONFIG.SYS in sysinit().

Under TACLI, configuration requires strict resource control: each TSR or driver loaded should be clear about what it owns, and the entire roadmap needs to be accessibly documented at all times. User programs should run only by accessing configured and documented system modules, whose heritage and resource ownership can be traced all the way back to boot, in parallel and without conflict with all other such modules.

This model is accessible in a very simple DOS-like system, in which the CONFIG.SYS is kept to a minimum, and the command prompt is treated as inside the box initialized by the AUTOEXEC.BAT, which in turn is inside the box initialized by the CONFIG.SYS. A minimal DOS-like OS is most desirable. Minimal specifications are more flexible analogically, and the instance system design should be applicable not only to the 8086, but also to modern chips with wide memory spaces, and to embedded systems which currently run as slaves without a command line.

For convenient development of the instance, we may start with an encapsulation of a standard operating system, such as a DOS subset, in a channel access kernel. Such code has already been demonstrated by my 1996 work, including encapsulation of standard input, standard output, and the printer. Neither the speed nor the range of capabilities of the channel-based program boxes is significantly affected.

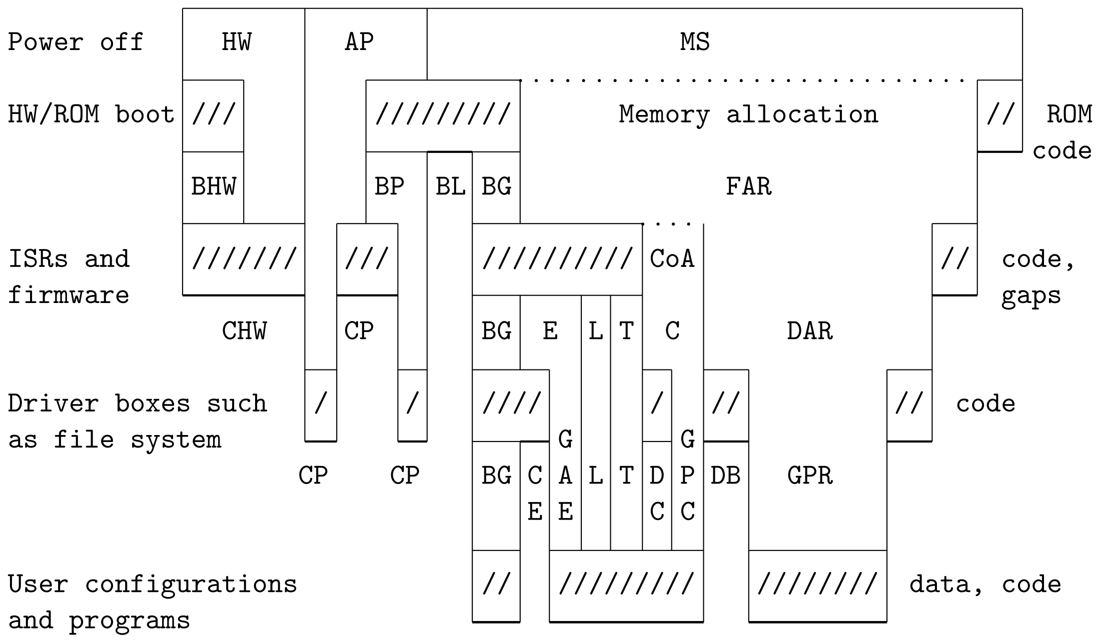
3.5 The exoskeleton

A partial snippet of the Transputer code for a DIO32 (parallel ports) TRAM is given below. Complex cases are similarly handled, including SCSI target/initiators.

```
PROC DIO32VCb (VAL INT invert.bits.from.PGM,
  CHAN OF ANY replyGraphics, inGraphics, outVelocity, inCruise)
  #USE "events.t2h"
  #USE "serial.t2h"
  -- Above four channels are all LINKS defined in enclosing .PGM file,
  -- but occam inconsistently does the PLACE of memory mapped ports
  -- within the .OCC code module.
  -- The AT addresses count two-byte words up from lowest
  -- The # addresses are in bytes from MOSTNEG INT = #8000
  INT WrPar0 :
  PLACE WrPar0 AT (#7800 >< (MOSTNEG INT)) >> 1 :
  [4]INT IntWr :
  PLACE IntWr AT (#7C00 >< (MOSTNEG INT)) >> 1 :
  [128]INT IndWrPar0 :
  PLACE IndWrPar0 AT (#7A00 >< (MOSTNEG INT)) >> 1 :
  -- and some other INTs... ending with...
  CHAN OF ANY event :
  PLACE event AT 8 :  -- hard "link" word location of event "channel"
  -- retypes of hardware specific declarations
  -- (this approach could allow all INT ports to be based on one PLACE)
  INT int.enable  RETYPES IntWr[0] :
  INT int.clear   RETYPES IntWr[1] :
  INT invert      RETYPES IntWr[2] :
  INT master      RETYPES IntWr[3] :
  -- hardware specific constants
  VAL INT int0             IS 1 :
  VAL INT int1             IS 2 :
  VAL INT int2             IS 4 :
  VAL INT int3             IS 8 :
  -- more declarations leading finally to...
  INT int.mask :
  SEQ
    int.mask := int0\/int2 -- power-up or reset, output bits stay low
    -- more code
:
```

This code performs the functions of a driver, yet it is nested outermost. It is the “harness” of Transputer lore [12], easier to maintain than a driver. That surprised Jim Sack [15], a colleague with wide non-Transputer experience. He was pleased with the name *exoskeleton:* the hard parts are on the outside.

The success of peripheral TRAMs is due to the resource-oriented paradigm implicitly followed using hand-coded PLACE statements. Despite faster hardware, competing standards like the Texas Instrument “TIM” were not as successful. They were hobbled by lack of the occam language, forcing either assembly coding or metaphor dangers (dynamic languages like C).



Legend:

HW	Hardware	AP	Addressable ports
MS	Memory spaces	BP	Boot ports
BHW	Boot hardware	BL	Boot-local RAM
BG	Boot-global RAM	FAR	Firmware-accessible RAM
CHW	Captured hardware	CP	Captured ports
E	Events	L	Hard links
T	Timers	CoA	Connector allocation
C	Connectors	DAR	Driver-accessible RAM
CE	Captured events	GAE	Generally accessible events
DC	Driver channels	DB	Driver-local buffers
GPC	General-purpose connectors	GPR	General-purpose RAM

Figure 2. BIOS-type exoskeleton: example diagram

Figure 2 sketches a generic exoskeleton, with BIOS features and an implicit command line (bottom). Resources extend horizontally, and nesting proceeds downward. An exoskeleton level can "capture" a resource by not passing it to lower levels, substituting specified regular usages. For instance, in the diagram, interrupt events replace hardware stimuli via soft-coded (non-Transputer) ISRs. Note that the topmost level, the configuration of multiple communicating CPUs, is omitted from Figure 2.

The array of global *connectors,* or single-word channel resources [5], is a TACLI feature. It is prefigured by soft configuration channels in ".PGM" files, and soft channels in harness code like DIO32VCb. It permits software drivers like file systems to be true, testable boxes. Connectors persist on a nesting level strictly enclosing those in which they are used [5], an occam feature that improved upon CSP [13].

TACLI will allow PLACE only at top (power-off) configuration level. It will be generalized to use the special port memory of the X86, in addition to dedicated maps of regular memory as shown here, and any other CPU-accessible digital hardware address space. Some of these addresses (like COM ports) will be relocatable and dealt with by the linker or even the loader. As with Transputers, the linker/loader will know to avoid the forbidden memory-mapped areas in code and workspace placement.

3.6 Filing and system IO including errors

The replacement of standard system IO libraries with communication libraries through channels may seem to be a challenge, but as a matter of fact it has been wholly solved. The old Inmos treasure, the occam toolset, provides IO header files and libraries like HOSTIO that operate through any level of process/channel interconnectedness and any topology to communicate with a non-occam host. These will work with no essential change to communicate with a resource-oriented host.

In the old Inmos toolset, compilers and servers had to run on the root Transputer, because DOS did not offer enough addressable memory to support the algorithms. Code on the OS-based PC side was essentially different from the channel-based Transputer side. TACLI eliminates this artificial barrier, and extends the "Transputer-like" domain all the way up to the OS or, for a first cut, its encapsulation. This concept was thoroughly proven in [5] and [6].

The Unix-like notion that "everything is a file" can be encapsulated if desired by "every file descriptor is a channel". Alternatively, more in the HOSTIO spirit, it can be dealt with through "every file descriptor uses a channel". Of critical importance in either case is the fact that by resource-oriented structure, this channel must be accessible at the outermost program level (since it is communicating with a module outside the program, namely the file system). Thus, tracing communication without significantly affecting performance is always easy.

Operating systems typically exercise centralized control in case of error. In TACLI, centralized control must be applied through a resource that is handed down through all nested code layers to its point of use. Due to the locally static nature of TACLI, and the complete definition of resource ownership and state at all times, a halt-on-error option like that of the Transputer will result in the stellar postmortem debugging performance characteristic of the Transputer — a far cry from nightmarish reset-button kernel debugging as currently practiced. Each postmortem is a return to boot, easily modeled by TACLI nesting.

The other option, branch-on-error, includes the option of halting only the erroneous process and allowing others to run till paused for communication, and is typical of production systems. The accessibility of formerly critical code in the box model means that diagnostic channels can penetrate to failure-prone places. They can also easily be added later. Diagnostic warning and restart capability can replace the hidden timeouts that so bedevil complex systems.

Hardware disconnection in the middle of a channel communication can be dealt with by techniques pioneered by Inmos in their "InputOrFail.c" family. These are based on a special channel reset instruction. A separate stop channel is needed which, as above, can easily be brought to the outside of the box, no matter how complex the code.

3.7 The conquest of the metaphors

Going through the metaphors of Section 2 one by one shows that the TACLI design strictly eliminates them. An examination of the strict design nesting of TACLI boxes will then show that this is sufficient to provide fully predictable behavior and, thus, that the proposed design satisfies the sufficient condition and is fully analogical.

The infinite metaphor, 2.1, is eliminated by the locally static nature of TACLI code, plus the requirement that resource ownership knowledge be fully accessible at all times. Even nested procedures are handled in a stackless fashion. Programs can still, of course, fail because of lack of resources, but knowledge of resource use is brought out to the outside of the box, permitting design control of such overflow. Since

TACLI code is complete, controlling all resources used by any program, the finiteness and design control is also complete.

The sequential metaphor, 2.2, is taken care of by the universal application of the PAR and channel constructs. This requires formerly critical communication between independent control streams to be brought out to the outermost layers of dependent boxes. The formerly critical kernel-type entities like interrupts, timers, DMA, task switching and IO states are modeled and controlled on the box boundary, using efficient stackless data handling, and easily accessible to top-level diagnostic code.

The call metaphor, 2.3, is flattened by moving the singular IO parts to the outside boundary of the box. Unless a channel formal is explicitly declared, calls are regular algorithmic. Calls with channel formals are also regular examples of a different analogy, one permitting pauses and state-based or resource-based flow, but they are not subject to side effects or ordering constraints dependent on distant code, since a channel in use is always, under control of an enclosing box, open.

The template object metaphor is eliminated because what takes the place of an object in TACLI is a combination of data template and complete box of controlling code. Since code is theory slavishly adhered to, this fully determines the behavior of the "TACLI object" under the stimuli provided from outside the box. There are no side effects or hidden stimuli. A TACLI box, as historically exemplified by a process run compiled by the "occonf" configurer, behaves equivalently whether it is implemented in hardware (multiprocessing) or software (multitasking). When used as a software component, it behaves just as an identically designed hardware component would. This includes error behavior, which always results in a fully defined state accessible from outside the box.

The architecture and design metaphor is solved by the nestability of real components, all of which are free of side effects, and are affected by, and produce, only transparently understood stimuli. Replacement of a component with an equivalent component, even if designed years apart and by a different team, will therefore produce a known and equivalent result. Outer design of a system, no matter how complex, can thus proceed with confidence based on a sequence of imposed conditions and resulting states.

Checks of "microcorrectness" no longer have to take the brunt of testing. True architecture, with real and demonstrable speed, latency and data capacity specifications, will take their place, just as in physical design. This redefines the notion of "software contracts" away from microchecks to genuine design specifications.

Designs of TACLI objects with multifaceted controls can be divided into multiple parallel boxes, each with a well-defined purpose, and communicating with one another. This naturally eliminates the need for spaghetti code or "aspect-oriented" programming. For instance, an allocation or security box can be in communication by channels with many substantive processes, including ones of different kinds, and designed at different times and by different teams. Channel overhead will be extremely low thanks to stacklessness, and if the "aspect" is a software box, the response will be nearly instantaneous.

Finally, hardware advances and diversification are no longer the strangling enemies encountered, for instance, by the air traffic control software porting project, which failed [2] after hundreds of millions of dollars and years of work. A TACLI box with channels is a universal encapsulating and middleware mechanism. It replaces the driver API and its instances, always slightly different in behavior, with a side-effect-free specification that strictly holds. Its quick response to stimuli is like that of kernel-based drivers, but it is subject to top-level controls, unlike buried drivers.

TACLI boxes, once designed and tested, will go on being useful forever, and not come to a grinding halt because of hardware, system, and technology changes in distant and uncontrollable sectors of the computing world.

4 Project plan

In order to compress the time expected before first language availability to around one year, I will limit the scope of initial development to one target system, CPU, and set of peripherals. All design will be done with the purpose of the initial instance being analogically expandable to cover any reasonable architecture. In particular, it will be kept critically compatible with the needs of the IBM Cell architecture.

4.1 Why this approach is easy

The structure targeted by the initial phase of the planned TACLI development is shown in Figure 3. It is designed specifically to minimize development time to a usable and exemplary product. The strategy is to stand on the shoulders of legacy giants who have given us two simple, powerful, and design-frozen systems: DOS and the Transputer.

Referencing Figure 3, the following features all combine to make development easy and robust:

4.1a. Platform stability: The initial phase will produce legitimate DOS program files and legitimate Transputer bootable (".BTL") images, running on a system consistent with both DOS and Transputer specifications, and therefore well known. Because of this consistency, well-known and powerful tools, owned by this researcher, will be usable: the MASM suite, CodeView, MEM.EXE or MI.COM (DOS); the toolset suite, check and mtest (Transputer).

4.1b. Powerful DOS default: Despite the caveat in [8] that DOS contains code that varies by "manufacturer", in fact a DOS floppy will boot a system other than the one on which it is made (with hard drive partition size restrictions). This means the DOS floppy starts with usable generic drivers, a great treasure, which are augmented by a simple but usable hierarchical file system (FAT) and easy hardware handling (everything is "real mode"). The DOS 3 - DOS 5 kernel is single-tasking, which is a further boon for us: it gets out of the way. The DOS memory map is clean and easily checked, and so is its hard interrupt behavior and port list.

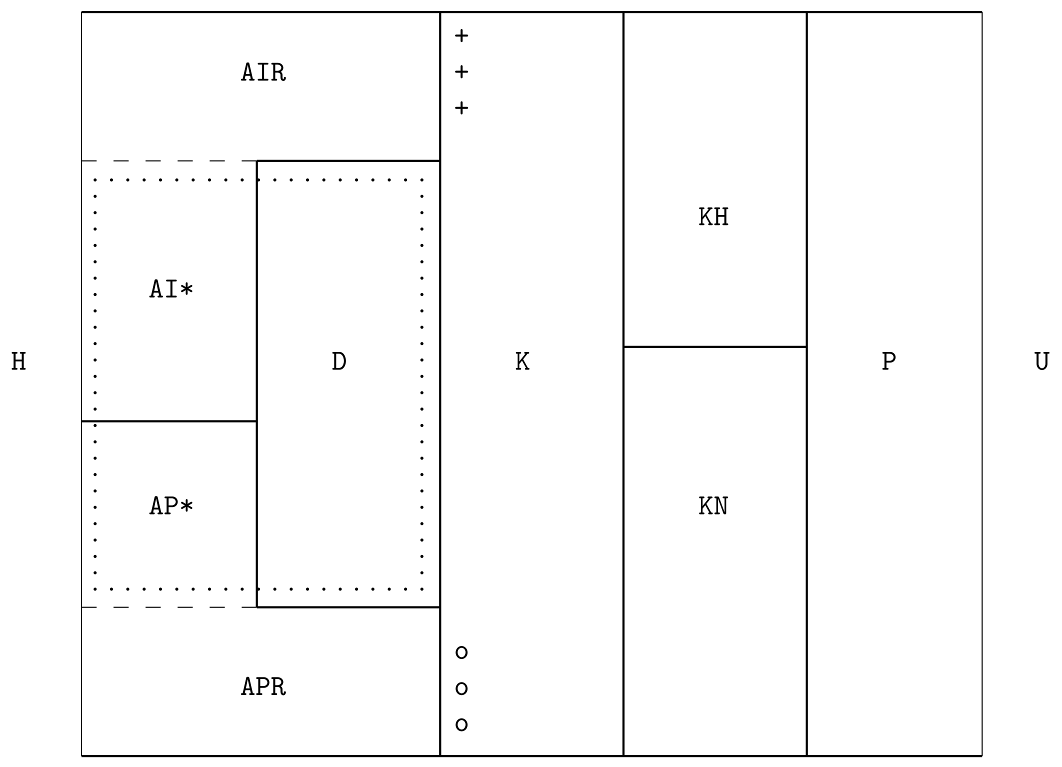
4.1c. Small, known specification: The full list of DOS soft interrupts is not large, nor is the boot specification for DOS or the Transputer, nor the HOSTIO library specification for Transputer channel IO, nor the description of DOS scripts (expanded by my daughter file specification). It is a subset of the union of these which must be supported — we can omit legacy features, like inner nested "PLACE" and "PLACED PAR" and the DOS File Control Block, because we do not have to support pre-1990 code. And the soft interrupts end up all hidden behind the kernel API, which controls which ones are ever used.

4.1d. Small kernel coding task: Figure 3 gives the mistaken impression that the asynchronous code (far left) is as big as the rest of DOS. Actually, it is far smaller, and what needs to be changed smaller yet. Once this is done, patterns already known from Transputer assembly code can be simulated. Except for a small amount of strict operating system work on PSPs and the like (already proven to work in [5]) this finishes the kernel.

4.1e. Clear compliance with "locally static" specifications: DOS soft interrupts have finite small stack depth and are not reentrant, which eases the task of controlling the depth of stack. The language design, in return, prevents runaway usages that might lead to deep hard interrupt nesting or other lack of resource control.

4.1f. Clear resource nesting model: The TRAM peripheral code examples, together with analogous PC peripheral code in assembly, yield robust and workable port addressing templates for the TACLI language.

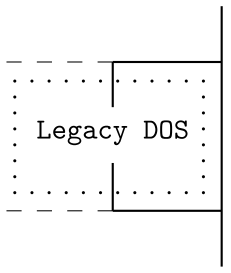
4.1g. Powerful modern compiler-writing tools: Once the language description is drawn up, we can use Linux and its array of free and powerful tools, like lex and yacc, to access many megabytes of memory in carrying out compiling and testing.



* NOTE: DOS code is left unchanged. Interrupts and port references are
redirected to code that has the same DOS response as DOS code.

Legend:

```
H   --- Peripheral hardware
AI  --- Hard interrupt code (DOS) including DMA done, port ready, timer
AIR --- Link and event response code (TACLI)
AP  --- Direct port access code (DOS) including memory-mapped IO
APR --- Port response code (TACLI)
D   --- DOS synchronous code (soft interrupts)
K   --- TACLI kernel
KH  --- TACLI channel API including HOSTIO-type libraries
KN  --- TACLI run API including tame/wild nesting and error response
P   --- TACLI/DOS application programs
U   --- User configuration and control
```



+ TACLI kernel link/event handling code, including timer

TACLI kernel calls to DOS soft interrupts

o TACLI kernel port handling code

Figure 3. TACLI over DOS operating system

4.1h. Easy debugging: Thanks to the static nature of the target code at any moment of time, our suite of debugging tools, such as CodeView, MI.COM, and the Transputer post-mortem debugger idebug, is capable of capturing exact state response, even dealing with things like DMA. This greatly benefited my 1996 development, and is not at all like the standard modern kernel-development nightmare.

4.1i. Full transparency: Once the link, port, and system channel libraries are developed, all system IO will be accessible at the top program level, and no longer buried deep in driver calls. This will allow scripted data flow testing of all programs.

4.2 Initial phase

Because of the large amount of work done previously on it, the fact that I already own its necessary software and hardware, and its simplicity and invariance, my likely development target will be the IBM-compatible PC-AT with 8086 and ISA bus specifications, connected to a DMA-capable Inmos B008 with several Inmos-compatible TRAMs. Peripherals will include DOS/FAT floppy drive and hard drive, Centronics printer with interrupt (HP Laserjet IIIp or similar), keyboard, and two legacy serial ports. The target system will be 16-bit, and the use of extended or expanded memory will be avoided.

```
   i. Analyze customary 1990-era programs for resource requirements.
      The programs will include screen editor, assembler, compilers,
      debuggers, multitasking program arrays (my 1996 demo HEART3), and
      both a DOS-like and a Unix-like command line. They will include
      floppy and hard drive IO, keyboard input, text display output,
      printer output, serial IO, DMA-based PC-Transputer IO, and
      TRAM peripheral code.

  ii. Define specific resource requirements, at the language and DOS
      soft and hard interrupt levels, to support (i). Define general
      resource requirements so as also to allow future support of
      mouse, graphics display, network IO, and RESETCH (link failure)
      and other error.

 iii. Define the APIs of KH and KN of Figure 3, using only the resources
      of (ii), and sufficient to support all the capabilities of (i).
      Iterate on (ii) and (iii) until converged.

 Kiv. Design the TACLI kernel K      Liv. Design scripting (outer) and
      and the asynchronous code           compilable (inner) language to
      AIR and APR to support it.          comply with (i)-(iii) and
                                          analogic extendability to
                                          modern CPUs and platforms.

  Kv. Code and test (Kiv).            Lv. Write scripter and compiler,
                                          and test on examples spanning
                                          the functionality of (i).

 Figure 4. Initial phase --- details
 -----------------------------------
```

The actual development of compiler and script tools will be done on a Linux system, which may share the same hardware as the PC-AT system. I will use standard GPL compiler creation tools. I will take advice from compiler writing experts on the syntax for use. Currently, I lean toward a C-like semicolon-terminated syntax as more familiar to most people, as opposed to an occam or Python-like line indent syntax. I also lean toward a scripting language syntax that, like Perl to C and the toolset configuration (".PGM" file) language to occam, bears a strong resemblance to the compiled language.

Development will be staged as shown in Figure 4 above.

Code structure in PC-AT memory space will be equivalent to a subset of DOS, including the standard program segment prefix. Code structure in Transputer memory space will be standard Inmos toolset processes, compatible with the occam2 beta release circa 1990 (reference 72 TDS 184 01) augmented by ASM assembly constructs.

In order to stand on the shoulders of legacy giants, I will begin by encapsulating standard DOS (between DOS 3 and DOS 5) in a small kernel that permits strict box/channel TACLI program runs from a command line. Similarly, I will start by translating TACLI programs to occam/ASM for the Transputer array, and compile, load and run them with the toolset. Boot control will begin with documentation of a standard, minimal DOS boot and configuration, and proceed to a script wrapper that produces predictable resource ownership configuration.

All peripherals will be scheduled through channels, and timer and event code will also be supported. Access in principle to total locally static resource ownership and state information will always be maintained. Pending the writing of an actual debugger that captures it, DOS and toolset debugging tools will suffice.

The TACLI model requires multitasking to be program-controlled, not OS-controlled. The DOS used will be single-tasking; all multitasking constructs will be my own daughter PSP handling, as mentioned in [5].

The deliverables of 4.2 will include detailed interrupt, DMA, and port state machine specification and code for each of the supported TACLI links and direct ports. It will include a language specification, compiler and scripter that complies with TACLI specifications, when run in conjunction with the DOS and B008/Transputer/toolset legacy tools. Also there will be tools to capture full ownership and resource state, within the restrictions I impose upon DOS use. These restrictions will not prevent normal use and efficiency of the CPU and peripheral capabilities of the system, which will be demonstrable when running TACLI programs.

4.3 Instance completion phase

As possible within the constraints of my time and funding, I will continue work on the development target system to clean up some of the compromises of 4.2. Except for the first paragraph, which is an absolute requirement, the following features are more a wish list and are subject to change.

The design of languages, ownership and resource state will be fully documented in an analogical way that is applicable to other systems, especially including the IBM Cell or similar devices. It should be as easy to extend TACLI to a new system as it is to extend C.

A fully TACLI boot script design and implementation should exist. This should include the capability of returning to boot state and a corresponding boot command line, functioning in a boot-level box that strictly encloses the equivalent of CONFIG.SYS, AUTOEXEC.BAT, and the standard command line.

Insofar as possible, DOS interrupt and driver code should be replaced with code designed for TACLI according to the documented analogical design. This eliminates the "D" box in Figure 3, replacing it with an extension of the kernel.

Transmission of bootable images (".BTL" files) to the Transputers without "iserver" should be implemented through a program driving the PC-to-B008 link directly as a TACLI channel.

TACLI designs for network, USB, and interrupt-driven mouse should be developed.

TACLI should support RESETCH and both halt-on-error and stop-on-error (in the Transputer sense) insofar as hardware allows. A TACLI postmortem debugger design should be developed, which respects the nesting level required for correct locally static access to the core state.

A TACLI design consistent with all of the above should be developed in detail for a second target system, unrelated to DOS.

5 References

1. Beck, Kent: *Test-Driven Development.* Addison-Wesley, Boston, 2003.
2. Boehm, Barry, and Richard Turner: *Balancing Agility and Discipline.* Addison-Wesley, Boston, 2003.
3. Dickson, Lawrence J: "Asynchronous PC occam firmware - progress report." email: tjoccam@tjoccam.com, 9 November 1993.
4. Dickson, Lawrence J: "Asynchronous PC occam firmware - progress report." email: tjoccam@tjoccam.com, 25 August 1995.
5. Dickson, Lawrence J: "occam (TM) Road Map for the DOS PC." *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA'96), Hamid R. Arabnia, editor. CSREA, Sunnyvale CA, 1996.
6. Dickson, Lawrence J: "Four-Part tjoccam Package TJOCCAMA-TJOCCAMD. - DOCUMENTATION with diskette." email: tjoccam@tjoccam.com. Copyright December 1996.
7. Dickson, Lawrence J: "Flat is Beautiful." *Proceedings of the 2004 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA'04), Hamid R. Arabnia, editor. CSREA, Las Vegas NV, 2004.
8. Duncan, Ray: *Advanced MS-DOS Programming, Second Edition.* Microsoft Press, Redmond, Washington, 1988.
9. Inmos Ltd: *IMS B008 User Guide and Reference Manual.* Inmos Document Number 72 TRN 223 00. Copyright 1990 Inmos, Ltd.
10. Inmos Ltd: *occam 2 Reference Manual.* Prentice Hall International Series in Computer Science, C. A. R. Hoare, editor. Prentice Hall, New York, 1988.
11. Inmos Ltd: *Transputer Instruction Set, a Compiler Writer's Guide.* Prentice Hall, New York, 1988.
12. Inmos Ltd: "Using the D705B occam toolset with non-occam applications." *Transputer Development and iq Systems Databook.* Inmos Document Number 72 TRN 219 00. Copyright 1989 Inmos, Ltd.
13. Inmos Ltd: "Communicating processes and occam" 2.5: "The alternative construct." *Transputer Applications Notebook: Architecture and Software.* Inmos Document Number 72 TRN 206 00. Copyright 1989 Inmos, Ltd.
14. Phoenix Technologies Ltd: *System BIOS for IBM PCs, Compatibles, and EISA computers, Second Edition.* Addison-Wesley, Reading, Massachusetts, 1991.
15. Sack, James G: Private communications, San Diego, 2006.