EasyChair Preprint

№ 5430

OCCAM-Equivalent Syntax with Pure Singleton Descent Structure

Lawrence Dickson

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

April 29, 2021

OCCAM-equivalent syntax with pure singleton descent structure

Lawrence J. Dickson
Space Sciences Corporation
Lemitar, New Mexico
Email: larry@spacesciencescorp.com

***Abstract*—Minor corrections and semantically valid restrictions, and one significant but equivalent change, allow the occam 2.0 syntax to be expressed in a yacc-compatible syntax (actually flex and bison are used) with very desirable properties. A rich 13-level descent structure (descent happens when a syntactic object is part of a definition of another syntactic object) is all singletons except for two levels (one equivalent to expression and the other equivalent to process), and these are broken up themselves into singletons by extended descent (parenthesis plus square bracket enclosure count, and indentation count, respectively). This will allow formal proofs of syntactic object behavior, and hence whole language behavior, to be vastly simplified.**

1. Introduction

During the 1970s and 1980s, a mathematical approach to computer language design led to the Communicating Sequential Processes (CSP) paradigm and the occam computer language. In parallel, the INMOS hardware development of the Transputer family took advantage of the fact that core features of CSP applied equivalently to hardware and software. The result was the classic CSP/occam principle of hardware-software equivalence (HSE), mentioned in [1] p 71: "Configuration does not affect the logical behavior of a program." This was successfully elaborated with a configuration language ([2] chapter 5). This means the design has many consequences in software harnessing and wrappers, as well as in embedded coding, which was a strong point of the classic Transputer. Unfortunately, after new ownership of INMOS, the custom hardware design toolkit was abandoned, and monolithic cost-effectiveness comparisons battered the Transputer, which fell victim to Moore's Law and faded with INMOS in the 1990s.

The fact that the practical consequences of Moore's Law are independent of, or even run counter to, mathematical elegance, does not imply that mathematical elegance should be abandoned in computing.

The area which is a natural fit to this CSP/occam/Transputer design is embedded programming, but in the 1990s both the hardware (T9000) and the language (occam-Pi) took a leap away from that toward dynamic, operating-system-type design targets. The T9000 died rather quickly, unable to compete with Moore's Law giants like Intel, while the "soft" occam descendants have struggled on to the present day. However, troubles with the maintenance of the "soft" occam toolsets have recently resulted in the revival of classic tool use, especially Gavin Crate's Transputer Emulator, in new projects such as the SME Transputer ([3], [4]).

The current author has always preferred static, embedded design, and wishes to bolster this rebirth with a rigorous language definition that will be able to support formal proofs and to expand in the direction of modern embedded programming and heterogeneous multiprocessing, including the Internet of Things (IoT).

2. Motivation

Any computing language is a mathematical definition. Any successful computing language is a mathematical definition that is applied to multiple special cases (programs). These special cases, at varying removes, end up touching the physical world via hardware, and their physical influences and effects are intended to be described by the mathematical consequences of the language.

In most of mathematics, a major definition with extensive consequences through many special cases is simple and well-defined. It does not change for decades. An example is a group, whose definition dates from 1854, and though vast work has been done on implications of this since then, the definition remains the same. Yet its consequences remain unexhausted by all this study.

Computing languages suffer "churning" by contrast, with even the most stable languages such as C getting decade by decade definition updates. As a consequence, the full implications of a language definition do not get explored before the crowds go thundering after newer and, usually, more complex definitions with changed parameters of emphasis. Algorithmic efficiency is the usual motive for the changes, and yet true mathematical proofs are rarely found, and surprises keep happening in apparently checked programs [5] due to hidden assumptions (like "no overflow") being violated.

In addition, most languages are practically unusable apart from an operating system (OS), which is a program, often not even written in the same language. OSs are typically even more complex than languages and more frequently changed. Formal verification has reached an OS only in

extremely rare cases like the formally verified microkernel, seL4 [6].

In this paper, we return to what is probably the solidest of useful computer languages from a mathematical point of view, and we define it very firmly using applicable classic reference manuals for the language [1], the toolset [2], and the Transputer [7]. Interestingly, the BNF found in [1] does not quite reach the level of correctness and rigor needed, although use of the toolset historically reached a very high level of reliability (the author's group rescued a Ford automotive radar project using this reliability).

Therefore, this author has proceeded under the presumption that adjusting the BNF to fit the semantics and practice should lead to a completely robust result. This presumption has not been disappointed, as this paper will show. The tools flex and bison (open-source lex and yacc variants) permit a complete language definition that breaks down into usable "building blocks" that avoid syntactic dependency loops, thus using only self-recursion and non-recursive building. Even more desirably, the self-recursion falls into two simple (and unavoidable) classes that will make resource and behavior theorems easy to prove step by step.

3. Singleton-friendly syntax adjustment and its implications for occam

In following sections, occlang7 and its demonstrable all-singleton syntax for occam 2.0 will be demonstrated. It is clear that, at least for that language, it brings formal verification within reach. This section will explore the meta-questions about why this is significant for CSP, occam, and the general direction of embedded computing.

3.1. Generalities

"Singleton-friendly" is not a complete description of the driving direction of this syntax, in fact it is rather a byproduct, a surprisingly simple outcome. To show the real roots, which have been there since the beginning — the author just cleaned up a few details — will make clear why the few syntactic adjustments were made, and why they "worked."

occam is CSP, but it is also *causal* and *realistic*.

Causality refers to the fact that progress in occam is triggered, and (in the Transputer implementation) triggered promptly. This is typically exemplified by the restriction to "input ALTs" (which could have been a restriction to "output ALTs": REQ-DTA instead of DTA-ACK). Attempts to allow both result in the Alphonse and Gaston "After You" impasse ([8] slide 3, [9], [10]). In fact every construct in occam allows prompt progress in the Transputer implementation, and hence is proven able to allow prompt progress in general.

Realism is rather subtle and sometimes taken for granted. Each occam construct can map onto real hardware or software (as in the Transputer) and *fully describe* what it is doing. Features of occam that feed into this include being static, being finite, using unbuffered channels, and being operating-system-free. Most languages give a partial description, and assume help from dynamic resource allocation, drivers, and other kinds of constructs at different layers of abstraction.

This goes really deep, to design causality. In C, for example, "language lawyers" insist that a union of different types may have undefined content, refusing to work from the known bit structure of each type. In occam, `RETYPES` takes the opposite view. The value of realism becomes quite clear with real projects. It is a main reason why workers in SME ([3]) moved from dynamically extended occam-Pi to Gavin Crate's legacy Transputer Emulator, which is pure occam 2.0.

3.2. Replacement of inline value process

The occam language [1] has a clear bottom-up structure, in which the expression, the operand, and the element are found within a line of code as in most languages, while the process (both sequential and parallel) culminates at a much higher level in block-structured fashion. Yet one feature, the inline value process, causes these two levels to be mixed, allowing a full many-line value process to be part of a single line of code.

It is unique even from the token point of view, once the full-line comments, preprocessor directives, empty lines, and continuation lines are removed, leaving only the INMOS syntactic NEWLINEs. The end of an inline value process is the only line whose first nonblank character is a close parenthesis. When that is not counted, the beginning of an inline value process is the only creator of parenthesis imbalance in a line.

It appears only in two syntactic entities, `expression_list` and `operand`, in exactly the same form in both. It has the same result(s) as a FUNCTION call instance, which permits its replacement by the ANY FUNCTION devised by the author. This is completed in occlang4.

As explained in detail in ANY-FUNCTION-construction.pdf in the GitHub ([20]), this results in no change in the semantics — or even in the two-dimensional text of the `value_process` enclosed in either form. Typically,

```
x ( value_process
  ) y
```

becomes

```
ANY FUNCTION name ( )
  value_process
:
x name() y
```

(where each of `x` and `y` are a number of entities).

It is always doable, because: (a) if two inline value processes intersect, then one is inside the other, so an inline value process depth can be defined, and innermost ones transformed first; (b) there is always a point above the line containing (the start of) the inline value process that has the

same variables in scope, and where an ANY FUNCTION or other specification can be inserted; (c) the number of possible names (about $52 \cdot 63^{n-1}$ names of length n) is so large that a unique new name can always be found.

The conversion to ANY FUNCTIONs has no semantic significance, since the value process is identical and the meaning of everything in scope is the same. However, its syntactic descent value is very great. It makes all indentations an even number of spaces, so that the tokens `RIND` (relative indent) and `ROUTD` (relative outdent) can be defined in place of large numbers of spaces at the beginning of each line. More important, it vastly enriches the descent structure, separating expression-like entities from process-like entities and clearly defining lines of code and multi-line entities.

3.3. Breakdown of types by categories

The other significant change going into occlang4 (other than an error fix, the special treatment of `andor_expression` and `andor_operator`) is the division of `primitive_type` into several subsets, `counting_type` (BYTE, INT, INT16, INT32, or INT64), `data_type` (BOOL, REAL32, REAL64, or `counting_type`), and `comms_type` (various CHAN, TIMER, and PORT types).

Certain definitions, such as the counted array protocol, are then narrowed, always according to the INMOS occam language semantics [1]. For example, the counted array protocol changes from

```
primitive.type :: [] type
```

to

```
counting_type DOUBLE_COLON INDEF_SPEC type
```

where `DOUBLE_COLON` means "::" and `INDEF_SPEC` means "[]" (token names), and internal periods are replaced with underscores so that yacc can deal with them. So the only real change is the restriction to `counting_type`, which is obviously semantically necessary, and will remain so in future definitions of occam.

The advantage gained from this (other than semantic precision) is that some of the new sub-`primitive_type` entities are plain combinations of tokens, and thus fall into a lower descent level (see sections below) than the "big one" equivalent to `expression`. This kind of disentanglement is key to the singleton structure achieved at the end.

3.4. Active and passive addressing and types

The changes between occlang4 and occlang6 are minor corrections clearly implied by the INMOS definition and the test suite (e.g. the fact that `"Hello"[3]` is legal). From occlang6 to occlang7 two major syntax adjustments are made, one a tightening and the other a loosening.

The tightening is a further development of `type` and its related entities. Though clearly valid for occam 2.0, its applicability to future developments of occam is not so clear, and requires some realism-based foundations. The discussion to follow is therefore very important, if not actually controversial.

Realism says that every resource in a digital computer is, in some way, addressed. There is a distinction between "conventional memory" and "everything else" which this author labels *passive addressing* and *active addressing*. This is clearly exemplified by the Intel 8086 memory address space and IO address space, but all computers must deal with it in some fashion.

In passive addressing, the state of the addressed area of memory, usually a multiple of a byte, is determined by a *write* and stays the same until the next write. Any *read* will yield only the value last written. No write or read has any side effects outside the addressed area. Of course, this is difficult to model in the modern era of cache, but it is typically a design and security goal.

In occam and CSP, with the three-fold conventional memory operations of assignment, input, and output, the "read" is phrased (e.g. in [1] pp 17, 54-55, 75) "used in expressions" and the "write" is phrased "assigned to by an input or assignment." Every read use including an output of the bare value involves an expression by language definition.

The type restrictions involving the new entity `non_comms_type` (which includes `data_type` and array types based on it) are meant to exclude every TYPE1 OF TYPE2 where both TYPE1 and TYPE2 are comms (PORT, TIMER, or CHAN). If we grant that all comms are active, the reason is that an active address has to have special hardware support, and its nature cannot simply be transmitted to an arbitrary target location by a communication. This is clear for PORT and TIMER, but might be disputed at least for a soft CHAN.

However, a basic CSP/occam characteristic is hardware-software equivalence, of which one form is the requirement that a CHAN formal must be agnostic as to whether it is a soft channel or a hard channel (link). Certainly, addressing a hard link has side effects, a serial transmission by wire, and so addressing a soft link must also be treated as active addressing. Its side effects are outside the process rather than outside the hardware device.

The upside of making these restrictions are that types and protocols, which formed a dependency loop in the expression-like level of syntactic objects, now resolve into extended descent. You build comms types either as TIMERs or as PORTs or CHANs of non-comms types. This works perfectly for occam 2.0, but we will later explore its implications for advanced language developments like mobile channels.

3.5. Value-only header at zero indentation

The second major adjustment from occlang6 to occlang7 is actually a correction. The original BNF in [1] did not concern itself with the full program, as required by yacc, but the author put that in, of necessity, in occlang4. It was done in simple library fashion (a set of PROCs or FUNCTIONs at indentation level 0).

This was not quite adequate, because real occam program files (.OCC files) usually insert an #INCLUDE file or files like `HOSTIO.INC` at the beginning. These consist of value name declarations and protocol declarations like `CHAN OF SP`. To accommodate this, `value_definition` was split away from `definition` to include PROTOCOLs and VAL RETYPES constructs, and `value_abbreviation` was split away from `abbreviation` to include VAL x IS y : constructs. Then a new entity, `value_header`, was built of `value_abbreviations` and `value_definitions`. And such an entity is allowed to lead the code in a `program`.

This works (example: the classic library `TIMES.OCC`). But it also has a subtle implication for future directions. It is a restriction to values, which are read-only, but includes PROTOCOL as a thing that might be shared across a library (which could be multiple separately compiled programs for different hardware components). This raises the question of sharing values between distant (hardware) processes.

The rule is that variables can be shared across a parallel if they are read-only but not if any member of the parallel writes to (changes) them. (See for example [1] pp 54-55.) Since channels are shared, this must mean the channel (its active nature, not the data sent over it) must remain unchanged over the connection lifetime. That model allows the idea of value sharing across a distance — it would require both sides to agree on what the values (including channels) are, and then not change them — which corresponds precisely to the "three-way handshake" used in standard TCP!

4. Mathematics

The purpose of this work is to define the occam language in such a way that mathematical proofs (formal verifications) are made comparatively easy. The language itself is defined using a "yacc-style" Backus-Naur Form (BNF) that runs successfully in bison, the open-source variant of the classic yacc language syntax builder. Given a lexical foundation (lexical objects: symbols, keywords, constants, and identifiers satisfying the requirements of a .L file, which gives regular expressions in a form usable by flex, the open-source variant of the classic lexical analyzer lex), this analyzes any legal program into syntactic objects (also called entities in this paper), according to a fixed finite list (.Y file) defining each syntactic object in terms of lexical objects and other syntactic objects.

The mathematical analysis of this section may be more easily understood if viewed in concert with examples from the next section (Section 5) and the full BNF listing in the Appendix.

The fact that syntactic objects are defined in terms of other syntactic objects induces a preorder [11]. This is useful because of the fact that the lexical objects, or tokens, are fully captured by a regular expression analysis before any syntax work needs to begin. This has been tested on a modified standard test suite originating from INMOS.

Definition 1 (Descent definition).
A *syntax universe* is a finite set *U* of unique *syntactic objects*, each of which has one or more *syntactic object definitions* in terms of other syntactic objects and lexical objects.
An *entity* is a syntactic object. A *full definition* is the set of all syntactic object definitions of an entity. An entity b is a *direct descendant* of an entity a if b is found on the right hand side of a syntactic object definition of a. c is a *descendant* or a *full descendant* of a if either $\mathrm{c = a}$, or for a finite sequence $\mathrm{b}_0 = \mathrm{a}, \mathrm{b}_1, \ldots, \mathrm{b}_n = \mathrm{c}$, where $n > 0$, b_i is a direct descendant of b_{i-1} for $0 < i \leq n$. The relationship between a and c is then called *descent*, and is denoted $\mathrm{c} \lesssim \mathrm{a}$.

It is worth noting that the equality case in the last paragraph can be included in the chain case by allowing $n = 0$, since then $\mathrm{a} = \mathrm{b}_0 = \mathrm{b}_n = \mathrm{c}$ and $0 < i \leq n$ never happens. The following is an easy consequence.

Lemma 1 (Descent preorder).
The descent relationship is a preorder.

Proof: The reflexive relationship $\mathrm{a} \lesssim \mathrm{a}$ is trivial. The transitive relationship ($\mathrm{a} \lesssim \mathrm{b}$ AND $\mathrm{b} \lesssim \mathrm{c}$ IMPLIES $\mathrm{a} \lesssim \mathrm{c}$) follows by chaining the b-to-a direct descents after the c-to-b direct descents, including the $n = 0$ case(s), QED.

Definition 2 (Equivalence classes and singletons).
If a and b are entities, we define *a is equivalent to b*, or $\mathrm{a} \sim \mathrm{b}$, if $\mathrm{a} \lesssim \mathrm{b}$ AND $\mathrm{b} \lesssim \mathrm{a}$. The *equivalence class of a*, or *A(a)*, is the set $\{\mathrm{b} \in \mathrm{U} : \mathrm{b} \sim \mathrm{a}\}$. Finally, a is said to be a *singleton* if its equivalence class *A(a)* is a singleton, that is, $\mathrm{b} \in \mathrm{A(a)}$ *IMPLIES* $\mathrm{b = a}$.

Lemma 2 (Descent partial order).
The descent relationship induces a partial order on its set of equivalence classes. If a and b are entities such that $\mathrm{a} \lesssim \mathrm{b}$, and A and B are their respective equivalence classes, then $\mathrm{c} \lesssim \mathrm{d}$ for every $\mathrm{c} \in \mathrm{A}$ and $\mathrm{d} \in \mathrm{B}$.

Proof: The second sentence follows from transitivity and the definition of the equivalence, and it implies the first sentence (see [11]), QED.

Definition 3 (Descent-closed sets).
A *descent-closed set* is a set *S* of entities such that $\mathrm{a} \in \mathrm{S}$ *AND* $\mathrm{b} \lesssim \mathrm{a}$ *IMPLIES* $\mathrm{b} \in \mathrm{S}$. If a is any entity, the *full descent set* of a is $\mathrm{D(a)} = \{\mathrm{b} \in \mathrm{U} : \mathrm{b} \lesssim \mathrm{a}\}$. The *full descent count* $c_D(\mathrm{a})$ of a is $|\mathrm{D(a)}|$.

Lemma 3 (Descent-closed unions). (A) A descent-closed set is the union of all the full descent sets of its members.
(B) The union of any number of descent-closed sets is a descent-closed set.
(C) The full descent set of any entity is a descent-closed set.
(D) If $\mathrm{b} \in \mathrm{D(a)}$ then exactly one of the following holds true:
(D1) $\mathrm{b} \sim \mathrm{a}$, *OR*
(D2) $c_D(\mathrm{b}) < c_D(\mathrm{a})$.

Proof: In the case of (A), clearly it must be a subset of that union, but every member of the union is reached by

descent from a member of the descent-closed set, which means it is also a member of that set. This proves (A). (B) follows trivially by definition, and (C) follows by transitivity. To prove (D), note that any two equivalent entities are descended from each other, and hence must have the same full descent set. Conversely, if $\mathrm{b} \in \mathrm{D(a)}$ but b is not equivalent to a, then a is not a member of D(b). Since D(a) is descent-closed, it follows that D(b) is a strict subset of D(a). But all these sets are finite, so (D2) is true, QED.

Definition 4 (Underset and next sets). The *underset* of an entity a is $\mathrm{D_u(a)} = \mathrm{D(a)} - \{\mathrm{a}\}$, the full descent set of a with a itself removed.
The *next step set* $\mathrm{N_u(S)}$ of a descent-closed set S is the union of all entities whose undersets are subsets of S.
The *next difference set* $\mathrm{N_d(S)} = \mathrm{N_u(S)} - \mathrm{S} = \mathrm{N_u(S)} \cap (\neg\mathrm{S})$, the set difference between the next step set and the descent-closed original set.

Lemma 4 (Next step is descent-closed). The next step set $\mathrm{N_u(S)}$ is the disjoint union of S and its next difference set $\mathrm{N_d(S)}$, and is descent-closed.

Proof: Because S is descent-closed, every member of S is a member of $\mathrm{N_u(S)}$. By definition, $\mathrm{N_d(S)}$ consists exactly of those members of $\mathrm{N_u(S)}$ which are not members of S. Therefore, for each member a of $\mathrm{N_d(S)}$, D(a) is the disjoint union of $\{\mathrm{a}\}$ and a subset of S. It follows that D(a) is a subset of $\mathrm{N_u(S)}$, and hence $\mathrm{N_u(S)} = \cup\{\mathrm{D(b)} : \mathrm{b} \in \mathrm{N_u(S)}\}$. Lemma 3(B) implies $\mathrm{N_u(S)}$ is descent-closed, QED.

Theorem 1 (All singletons in next difference set). Every member of a next difference set is a singleton.

Proof: Let a be a member of the next difference set for the descent-closed set S, and assume $\mathrm{a} \sim \mathrm{b}$. Then by definition of the next difference set, $\mathrm{D(a)} \cap (\neg\mathrm{S}) = \{\mathrm{a}\}$. Since $\mathrm{D(a)} = \mathrm{D(b)}$, it follows that $\mathrm{D(b)} \cap (\neg\mathrm{S}) = \{\mathrm{a}\}$. Since S is descent-closed, b is not a member of S, otherwise a would also be a member. Since $\mathrm{b} \in \mathrm{D(b)} = \mathrm{D(a)}$, it follows that $\mathrm{b} \in (\mathrm{D(a)} \cap (\neg\mathrm{S}))$, hence $\mathrm{b} = \mathrm{a}$, QED.

It is important to note that $\mathrm{N_d(S)}$ *may be empty.* Theorem 1 thus gives a technique for finding singletons. Start with any descent-closed set S, which may or may not consist wholly of singletons. Letting $\mathrm{S_0} = \mathrm{S}$, keep expanding it by defining $\mathrm{S}_i = \mathrm{N_u}(\mathrm{S}_{i-1})$, until you reach a value of i such that $\mathrm{S}_{i+1} = \mathrm{S}_i$. If S_i equals the whole universe, you are done; otherwise augment S_i by an appropriate descent-closed set containing non-singleton members. Ideally, every new member added to S_i by this process, thus giving an augmented S_{i+1}, has a full descent set which is the union of its equivalence class and a subset of S_i (see Lemma 3(D)).

This proves to work with the author's occam 2.0 syntax universes, occlang6 and occlang7. There are two places where augmentation has to take place, and each one involves adding only one non-singleton equivalence class. The first is the equivalence class containing expression and operand, and the second is the equivalence class containing process. The following Theorem eases the task of proving this.

Theorem 2 (Augmenting certain descent-closed sets). Suppose N is a positive integer, and we define $\mathrm{S}_{<N} = \{\mathrm{a} \in \mathrm{U} : c_D(\mathrm{a}) < N\}$. Suppose, in addition, that E is an equivalence class, or a union of equivalence classes, such that for any $\mathrm{b} \in \mathrm{E}$ we have $c_D(\mathrm{b}) = N$. Then both $\mathrm{S}_{<N}$ and $\mathrm{E} \cup \mathrm{S}_{<N}$ are descent-closed.

Proof: If $\mathrm{b} \in \mathrm{S}_{<N}$, then Lemma 3(D) implies any descendant of b has a full descent count less than or equal to that of b, and hence is also in $\mathrm{S}_{<N}$. In addition, Lemma 3(D) implies any descendant of a member of E is either a member of E or has a full descent count less than N. This implies both $\mathrm{S}_{<N}$ and $\mathrm{E} \cup \mathrm{S}_{<N}$ are descent-closed, QED.

Corollary 1 (Descent count step). Suppose N is a positive integer, and we define $\mathrm{S}_{<N} = \{\mathrm{a} \in \mathrm{U} : c_D(\mathrm{a}) < N\}$. Suppose, in addition, that $\mathrm{E} = \{\mathrm{a} \in \mathrm{U} : c_D(\mathrm{a}) = N\}$. Then both $\mathrm{S}_{<N}$ and $\mathrm{E} \cup \mathrm{S}_{<N}$ are descent-closed.

Proof: The only change in the hypotheses between the Theorem and the Corollary is the hypothesis on E. However, by definition, all members of an equivalence class must have the same full descent set, and hence the same full descent count. It follows that E of the Corollary contains the equivalence class of any member of E, and hence is a union of a number of equivalence classes, so the hypothesis of the Theorem holds. This also follows directly because $\mathrm{E} \cup \mathrm{S}_{<N} = \mathrm{S}_{<(N+1)}$, QED.

5. Modeling and reduction to singletons

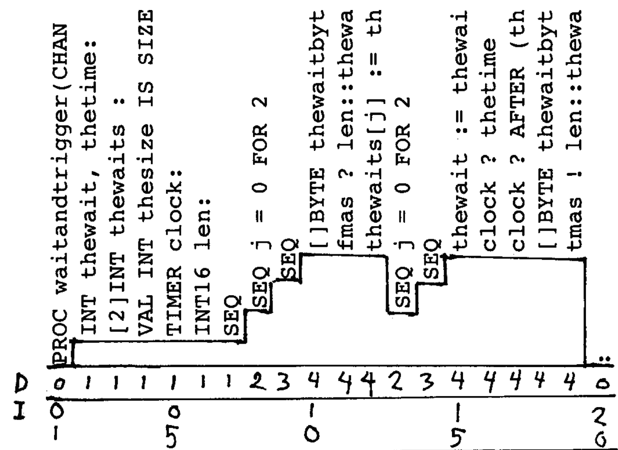

Figure 1. Indentation depth pattern of a real occam program (childcs.occ)

The full language syntax work is found ([19]) in ORG3 = https://github.com/SpaceSciencesCorp/Hard-Stuff-Language/tree/master/occorg3 in subdirectories langname = `occlang6` and `occlang7`. If ORGLANG = ORG3/langname, then the sorted full descent set for each entity *synobjname* is found in the file ORGLANG/srt9/*synobjname*.srt.

In both syntaxes, we start with S_0 = the set of all entities with no direct descendants. (These are the syntactic objects whose definitions include tokens alone.) Each of these has only itself as a full descendant, and therefore S_0 is descent-closed and consists only of singletons.

In both cases, the singleton-finding technique described after Theorem 1 leads to S_2, which proves to be the set $\mathrm{S}_{<N_3}$ in the following table. Augmenting as suggested in Corollary 1 for $N = N_3$ yields S_3, and more of the singleton-finding technique gets as far as S_9, which proves to be the set $\mathrm{S}_{<N_{10}}$. Augmenting as suggested in Corollary 1 for $N = N_{10}$ yields S_{10}, and two more singleton-finding steps consume the whole universe.

TABLE 1. COROLLARY 1 STRUCTURE

	total	N_3	N_{10}	Number of entities with full descent count			
				$< N_3$	$= N_3$	$< N_{10}$	$= N_{10}$
occlang6	99	29	97	15	16	68	29
occlang7	102	28	99	15	15	71	29

To explain some values in Table 1, notice that some of the members of $\mathrm{S}_{<N}$ may not be in the full descent set of any member of $\mathrm{S}_{=N}$.

All that remains after Table 1 is the question of how complex the two E-sets $\{\mathrm{a} \in \mathrm{U} : c_D(\mathrm{a}) = N\}$ are for $N = N_3$ and $N = N_{10}$. As can be checked by diffing members of ORGLANG/srt9, each consists of a single equivalence class. $\mathrm{E}_3 = \{\mathrm{a} \in \mathrm{U} : c_D(\mathrm{a}) = N_3\}$ is the set of entities equivalent to expression or operand, while $\mathrm{E}_{10} = \{\mathrm{a} \in \mathrm{U} : c_D(\mathrm{a}) = N_{10}\}$ is the set of entities equivalent to process.

To reduce E_3 and E_{10} to singletons, *extended descent* is used. In this, the E-set takes the place of the universe, treating lower level entities as tokens, since they can be completely defined before any entity of the given level or higher is dealt with. In addition, each E-level entity is *indexed* by a *depth* based on a globally defined count that is lexically accessible.

Newlines and indentation have syntactic significance in occam. By replacing inline value processes with ANY FUNCTIONs, all syntactically significant indentations are a multiple of two spaces, and the tokens RIND (relative two-space indent) and ROUTD (relative two-space outdent) are introduced to control these. Full-line comments, preprocessor directives, and empty lines are deleted before syntactic analysis begins, and continuation lines are merged into one line. *Syntactic newlines* (a subset of *lexical newlines*, excluding full-line comments, preprocessor directives, empty lines, and continuation lines) get the token NEWLINE. A *line positioning* is either the beginning of file or a NEWLINE followed by one RIND or by zero or more ROUTD, and a *full line positioning* is a line positioning not followed by NEWLINE, RIND, or ROUTD (i.e. it is either end-of-program or is followed by something other than NEWLINE, RIND, or ROUTD).

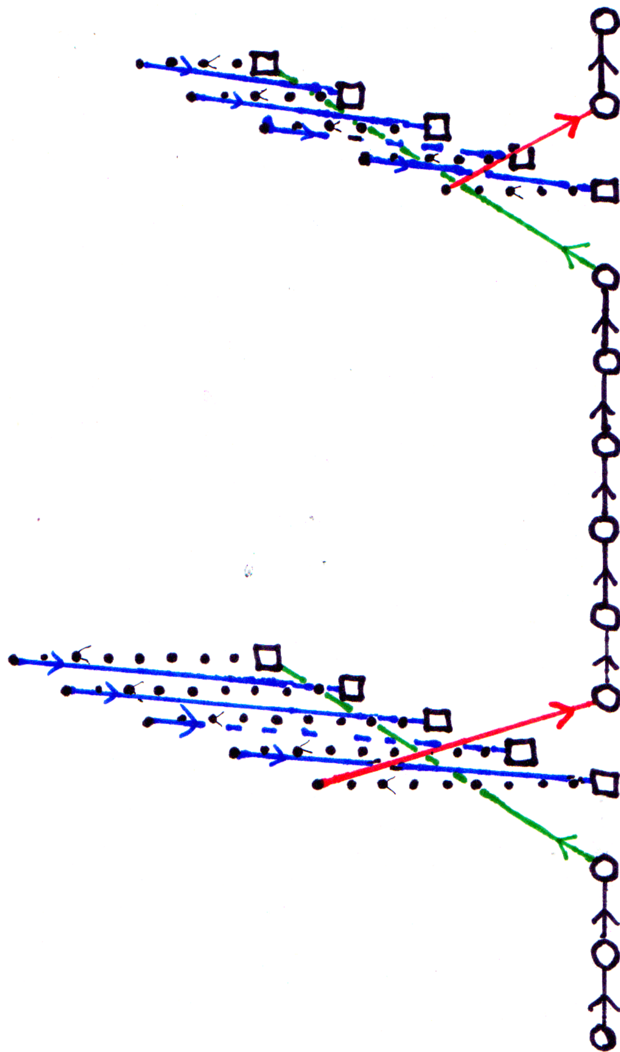

Figure 2. Ascent pattern for occlang7, using extended descent

A *syntactic line* begins after a full line positioning and ends with the next NEWLINE. A *multi-line syntactic entity* begins with a syntactic line and may contain further syntactic lines, RIND tokens, and/or ROUTD tokens, but must end at the same indentation at which it began, without ever internally reaching a lesser indentation. A *sub-line entity* is an entity that is neither a syntactic line nor a multi-line syntactic entity.

In occlang6 or occlang7, every entity that directly or indirectly contains a NEWLINE is a syntactic line or a multi-line syntactic entity. Every legal occam program begins and ends with a syntactic line that is at indentation depth 0. (Compare Figure 1.)

In E_3 the depth of an entity (which is always a sub-line entity in this case) is its *enclosure count*, the number of open parentheses and open square brackets minus the number of close parentheses and close square brackets before it on its

syntactic line. In E_{10} the depth of an entity (which is always a full-line or multi-line entity in this case) is its *indentation count*, the number of RIND minus the number of ROUTD before it in the file. The syntax guarantees that these do not go negative at any point in a legal program.

In each reduced universe, any entity is assumed to be *tagged* with its depth, and greater depths are fully evaluated before lesser. (This is always possible because a legal program is finite.) This means that, in a syntactic definition, any relative positive enclosure or indentation (as appropriate) implies that the enclosed or indented entities are already defined and may be treated as tokens. This is accomplished in the algorithms by changing their names so they do not correspond to names of real syntactic objects (the algorithms then treat them as if they were keywords).

A final variant must be treated for completeness. It turns out in occam that no syntactic definition in a reduced universe ever increases enclosure or indentation (as appropriate) by more than 1. This means there is one special case, the leaf, where the entity in question is depth-maximal in the program. It must therefore use a definition that has no added depth required. Some entities disappear because none of their definitions satisfy this; others disappear because they are dependent on disappeared entities, or because all that is left is an infinite regress definition. After this reducing process is complete, a special reduced universe for the leaf remains, which in turn is subjected to the algorithms.

Application of the algorithms (the singleton finding described at the top of this section) for syntax occlang7 proves that all four cases break down completely into singletons. (Compare Figure 2.)

The standard descents, as explicated by the approach of Table 1, are shown in Table 2. This includes every entity of occlang7, and except for levels 3 and 10, they are all singletons.

The extended descent required for level 3 is shown by Table 3 for the regular branch, and Table 4 for the leaf. This is visible in Figure 2 expansion of level 3, as the far ladder, to which the green arrow leads, has one less rung than the others.

The extended leaf levels look the same at first glance as the extended regular branch levels, but an examination of a single syntactic object shows how they get the given levels. On the standard CRI.nl, `element` has the definitions

```
element = element '[' subscript ']'
| '[' element FROM subscript FOR count ']'
| scalar
```

In the reduced syntax for level 3, general branch case, we have

```
element = element '[' subscript1 ']'
| '[' element1 FROM subscript1 FOR count1 ']'
| scalar
```

And in the even more reduced syntax for level 3, leaf case, we have

```
element = scalar
```

TABLE 2. STANDARD LEVELS OF ALL ENTITIES OF OCCLANG7

Level	occlang7 entities	
0	andor_operator counting_type integer name	byte dyadic_operator monadic_operator real
1	data_type scalar	name_clist string_head
2	data_type_clist string	literal
3	andor_expression conversion element expression_clist non_comms_type primitive_type simple_protocol table	comms_type count expression expression_olist operand protocol subscript
4	actual base case_expression expression_list selector type	allocation boolean channel output_item simple_protocol_slist variable
5	actual_clist declaration indef_type output_item_slist sequential_protocol	case_expression_clist delayed_input input_item replicator variable_clist
6	actual_olist input_item_slist specifier	assignment output tagged_protocol
7	formal tagged_list value_abbreviation	instance tagged_protocol_vlist
8	abbreviation input	formal_clist value_definition
9	action guard	formal_olist value_header
10	alternation alternative_vlist case_input choice_vlist construction function_body guarded_choice option parallel placedpar_vlist process selection specification value_process variant_vlist	alternative block_definition choice conditional definition guarded_alternative loop option_vlist placedpar procedure_body process_vlist sequence valof variant
11	block_definition_list	
12	program	

Thus, in branch case, `element` belongs in extended level 1 because of a self-recursion, while in leaf case, it references only standard level less than 3, and so is in extended level 0.

TABLE 3. EXTENDED LEVELS OF ALL OCCLANG7 LEVEL 3 ENTITIES

Level	occlang7 level 3		
0			
1	element	non_comms_type	table
2	operand	simple_protocol	
3	andor_expression	protocol	
4	comms_type		
5	primitive_type		
6	conversion		
7	expression		
8	count	expression_clist	subscript
9	expression_olist		

TABLE 4. EXTENDED LEVELS OF ALL OCCLANG7 LEVEL 3 LEAF ENTITIES

Level	occlang7 level 3 leaf		
0	element	non_comms_type	table
1	operand	simple_protocol	
2	andor_expression	protocol	
3	comms_type		
4	primitive_type		
5	conversion		
6	expression		
7	count	expression_clist	subscript
8	expression_olist		

Similarly, the extended descent required for level 10 is shown by Table 5 for the regular branch, and Table 6 for the leaf. This is visible in Figure 2 expansion of level 10, as the far ladder, to which the green arrow leads, also has one less rung than the others.

TABLE 5. EXTENDED LEVELS OF ALL OCCLANG7 LEVEL 10 ENTITIES

Level	occlang7 level 10		
0	alternation conditional loop sequence	block_definition guarded_alternative placedpar	case_input guarded_choice selection
1	definition	parallel	placedpar_vlist
2	construction	specification	
3	alternative process	choice valof	option variant
4	alternative_vlist procedure_body variant_vlist	choice_vlist process_vlist	option_vlist value_process
5	function_body		

The conclusion, that ascent shown in Figure 2 is valid for every program, is central to this paper: *if the syntactic objects of Level 3 are depth-tagged by enclosure count, and the syntactic objects of Level 10 are depth-tagged by*

TABLE 6. EXTENDED LEVELS OF ALL OCCLANG7 LEVEL 10 LEAF ENTITIES

Level	occlang7 level 10 leaf		
0	alternation definition sequence	case_input placedpar	conditional selection
1	parallel	placedpar_vlist	specification
2	alternative	choice	construction
3	alternative_vlist	choice_vlist	process
4	procedure_body	process_vlist	

indentation count, then the entire occlang7 syntax consists of singletons only. Each syntactic object definition is either a non-recursive *build*, a finite fixed sequence of lexical tokens and lower-descent entities, or it is a *self-recursion*, a finite fixed sequence of lexical tokens, lower-descent entities, and itself.

6. Complete details on occlang7 self-recursions

The use of occlang7 ascent to analyze the mathematical consequences of any occlang7-syntax occam program has now been simplified in principle to the stepwise analysis of two kinds of syntactic object definitions: builds and self-recursions. Each build is in principle trivial: the tokens and lower-descent entities and their sequence are known, explicitly listed in the definition of the build, and therefore as direct consequence the build is known (where "known" refers to timing, resources, and states before and after).

This leaves the self-recursions. A simple search of all the syntactic object definitions in occlang7 will find them all. There are 102 syntactic objects and 271 syntactic object definitions in occlang7 (see `alldefs.txt` in the GitHub ([19]), or look at the grammar in `y.output` there, after running the scripts). The search for self-recursions is trivial (same entity on left-hand-side and right-hand-side) and yields 38 self-recursive definitions (`occlang7-recurse-3-10.txt` and `occlang7-recurse-not-3-10.txt` show the 21 that are in Levels 3 and 10, and the 17 that are not, respectively). Three are discarded from the first list because from the extended descent point of view they are builds:

```
element = '[' element FROM subscript FOR count ']'
placedpar = PLACED PAR replicator NEWLINE RIND placedpar ROUTD
table = '[' table FROM subscript FOR count ']'
```

This leaves a total of 35, of which 18 are from Levels 3 and 10, and 17 are not.

We will completely analyze these 35. It turns out that they fall into two categories: *lists* which append items to the right or below, and *determinations* which insert items to the left or above. There are 24 of the former and 11 of the latter. They mostly fall into repetitive categories, as shown by Table 7 and Table 8 respectively.

TABLE 7. OCCLANG7 SELF-RECURSIONS: LISTS

Template	*item*	
item_clist = ***item***_clist ',' ***item***	actual data_type formal variable	case_expression expression name
item_slist = ***item***_slist ';' ***item***	input_item simple_protocol	output_item
item_vlist = ***item***_vlist ***item***	alternative option process variant	choice placedpar tagged_protocol
item = ***item*** '[' subscript ']'	element	table
Special cases		
andor_expression = andor_expression andor_operator operand block_definition_list = block_definition_list block_definition string_head = string_head STRING_LITERAL_MIDDLE value_header = value_header value_abbreviation value_header = value_header value_definition		

TABLE 8. OCCLANG7 SELF-RECURSIONS: DETERMINATIONS

Template	*item*	
item = specification ***item***	alternative option valof	choice process variant
item = allocation ***item***	process	
item = '[' expression ']' ***item***	indef_type type	non_comms_type
Special case		
indef_type = INDEF_SPEC indef_type		

7. Progressive details on syntax adjustments

The occlang7 syntax was developed over time through other syntaxes, of which we here reference occlang4 through occlang6: the earlier ones were consistent but not complete, and did not have the all-singleton property shown in Figure 2. This section gives specific details, that can be checked in the GitHub ([19]), not only of the generic descriptive changes described in Section 3, but also other changes including corrections both in the syntax of [1] and in the author's work. In the column labeled "Fix", e refers to an error in the original INMOS BNF, and o refers to an omission in the author's previous syntax.

Descriptions of items in Table 9:

A: Lexical foundation: These changes move as much as possible to regular expressions handled by lex (flex). These include literals including four varieties of string literals, three of which serve the multi-line string ([1] 3.2, p 26). `name` and `string` are no longer token names, but syntactic objects.

B: This is the type breakdown of Subsection 3.3.

C: This is the conversion of inline value processes to ANY FUNCTIONs, described in Subsection 3.2.

D: Parentheses may be omitted in strings of boolean operators ([1] 7.2.6, p 48), which is incorrectly omitted from the INMOS BNF. Here AND and OR may be mixed, and the order is left to right; not documented, but compilation experimentation shows it, e.g. for FALSE AND FALSE OR TRUE.

TABLE 9. CHANGES BETWEEN [1] SYNTAX AND OCCLANG4

Item	*Syntactic objects*		*Lexical objects*		*Fix*
	Add	*Remove*			
A	dyadic_operator name monadic_operator string_head string	digit exponent hex_digit	*Remove* name *Add* BYTE_CONSTANT INT_CONSTANT IDENTIFIER	 string HEX_CONSTANT REAL_CONSTANT STRING_LITERAL*	
B	comms_type counting_type data_type				
C					
D	andor_expression andor_operator				e
E	block_definition block_definition_list program				e
F	*_clist *_slist *_vlist	variable_list	NEWLINE ROUTD	RIND	
G	indef_type				
H	scalar	tag			
I		port timer timer_input			

E: Top level constructs, with `program` at the apex, are required by yacc (bison). This satisfies the requirement, using the format of a zero-indentation library of PROCs or FUNCTIONSs.

F: Many organizational changes with no syntactic/semantic implications are required by yacc. Two-character symbols must be named, curly bracket expressions must be eliminated and named and built recursively, and the use of tokens `NEWLINE`, `RIND`, and `ROUTD` is introduced as described in Subsection 3.2.

G: The new entity `indef_type` is introduced to break up `specifier` so that the occurrence of `INDEF_SPEC` (the digraph [], meaning an unspecified dimension) is within the new entity only.

H: Due to the exigencies of bison, it is convenient to combine `tag` with that use of `name` which names a variable. This is called `scalar`, a slight misnomer (it can be an array name).

I: `port` and `timer` are combined with `channel` due to their syntactic similarity in use. They will be distinguished by type declarations. They are actually closely related, as active addresses (see Subsection 3.4).

TABLE 10. CHANGES BETWEEN OCCLANG4 AND OCCLANG5

Item	*Syntactic objects*		*Lexical objects*	*Fix*
	Add	*Remove*		
J				o

Descriptions of items in Table 10:

J: The author's occlang4 omitted the `NEWLINE` after the definition of `tagged_protocol` that is `scalar` (i.e. tag) by itself on a line. This was consistent, but incorrect.

Descriptions of items in Table 11:

K: The author neglected to require escaping `'` in a `BYTE_CONSTANT`, and to require escaping `"` in the four

TABLE 11. CHANGES BETWEEN OCCLANG5 AND OCCLANG6

Item	*Lexical objects*	*Fix*
K	*Revised* BYTE_CONSTANT STRING_LITERAL*	o
L		o
M		e

STRING_LITERAL* varieties. This is corrected in occlang6.

L: The author omitted equality (the symbol =) from the list of dyadic operators. This was consistent but incomplete, and is here corrected.

M: A `string` must be a `table`, not a `literal` as the INMOS BNF claims, to allow expressions like `"Hello"[3]`, as the test programs do.

TABLE 12. CHANGES BETWEEN OCCLANG6 AND OCCLANG7

Item	*Syntactic objects*		*Fix*
	Add	*Remove*	
N	value_abbreviation value_definition value_header		e
O			e
P	non_comms_type	array_type	

Descriptions of items in Table 12:

N: As described in Subsection 3.5, an additional breakup of `definition` and `abbreviation` can allow a `value_header` to be defined that, like `HOSTIO.INC`, may precede all the zero-indent members of a library.

O: Inconsistently, the INMOS BNF makes the FOR value a `count` for a `table` but a `subscript` for an `element`. Though there is a hint in the names of Transputer assembly instructions *csub0* and *ccnt1* that one might allow 0 and one not, [1] 6.3 p 41 and 7.1 p 44 makes it clear there is no distinction.

P: As described in detail in Subsection 3.4, it is appropriate to introduce `non_comms_type` to unwind the possible comms types. Also, `array_type` proves to be unnecessary, and eliminates the last recursive loop.

8. Conclusions and implications for future static occam development

The occam 2.0 language had an impressive run, in the late 1980s and early 1990s, at the head of embedded programming — including massively parallel and heterogeneous hardware with all kinds of then-available IO (e.g., [12]) — despite a paucity of hardware for it to run on. It was always respected for extremely reliable and predictable program behavior, verging on the verifiable. This paper covers the last few gaps that must be covered to bring formal verifiability within reach. The consequences of this are considerable, and justify "picking up where we left off" to bring this technology forward.

8.1. Verifiability

Formal verification is nothing other than mathematical proof of physical consequences, applied to computing. CSP [13] gets us part way there, but current tools like Failures-Divergences Refinement (FDR) are too small, requiring exploration of all data cases separately using a functional language, and thus unable to deal with practical-sized problems. But nested mathematical proofs are valid if all their parts are valid. This allows a procedural language like occam to have provable consequences if it is causal, as occam is, and realistic with fully describable consequences, as occam is (see 3.1).

Most languages only partially describe reality (compare the full description in a math function like `z=atan2(x,y)` with the implied OS and interrupt support of its lookalike, `z=read(x,y)`). Given occam's full grip, all the way down to IO addresses and interrupts, plus hardware that fits its requirements, including side-effect freedom, all that remains is the nested steps (Figure 2). What is demonstrated in this paper starts with the token primitives (compare A in Table 9) and gets all the way to a program, possibly running on multiple heterogeneous hardware components. The beauty of it is that the singleton structure proved here suffices to show that all the steps needed are just those already described fully in occam and Transputer documentation such as [1], [7], and [2].

8.2. Development paths

Work has already been done on the denotational semantics of occam1 ([16]) and occam2 ([18]), and on large numbers of transformation laws valid in occam ([17]). These have sufficed to perform several practical projects, including the Transputer floating-point unit. However, they have been restricted by state space questions, for instance in WHILE loops. The current paper with its nested mathematics shows paths forward on this.

This leaves the question of how to move forward from around 1990 to the present computing world. At the level of occam foundations, which is OS-free and hence not subject to "churning," things are comparatively simple (like an Arduino rather than a Raspberry-Pi). But there are still thirty years of developments to deal with.

8.2.1. Current static advancements. Some of the advancements have already been done. The occam-like configuration language ([2] Chapter 5) is a clean path to heterogeneity, and most of occam 2.5, like records (the equivalent of C structs), is a reasonable simplification. The only exception may be the narrowing of the counted array protocol, but that can be avoided simply by sticking to the old standard.

8.2.2. Mobiles. The dynamic occam extensions such as occam-Pi [14] have, in this author's opinion, lost the thread by losing HSE, and yet some of the problems they tackled need to be dealt with but in a static fashion. These include mobile channel ends (functionally like TCP) and mobile

arrays (functionally like block devices). Work has already started, such as WACOMSAS [15] and the attachable ([8] slides 23-25) for the TCP-like development, and the detachable ([8] slides 21-22) for the block-device-like development. The attachable goes beyond the mobile channel end by using the shared-value-at-a-distance concept (see 3.5) and static constructs. The detachable replaces copies with moves in certain channel communications. Both carefully preserve HSE.

8.2.3. Hardware. The Transputer, though a pioneer chip for massively parallel design, was itself a CPU uniprocessor. Driven by the collapse of the cycle-time side of Moore's Law, modern dies have gone far beyond that, both in width (multiprocessing) and depth (cache). Work has begun here too, using an occam-to-Go language translation ([8] slides 6-15) as a template, since Go must deal with both these (and all occam lexical objects prove to be expressible in Go). Riding herd on cache, though in general very difficult, may be simplified considerably by the static and verifiably nested nature of occam, which was already friendly in the 1980s to the Transputer's "cache", its on-chip memory.

Another path into hardware that is closely allied with this paper is SME, which uses a related subset of CSP to vastly simplify FPGA design, including the emulated T425 itself ([4], [3]). This synergy was long ago demonstrated by the Transputer floating point unit design. Mathematical support for all these directions has begun ([9], [10]).

8.3. Conclusion

Simplicity is the path forward, not complexity. Massive and heterogeneous parallelism is the friend of clarity in computing, not its enemy. The reason for this is that it leads to understandability of individual functionality, like a village with paths between the shops of all the special craftsmen. The best computing of the future will be done following the embedded model, not the massive OS model. And it will all be formally verifiable, using mathematical proofs, and the simple nested structure implied by the singletons of this paper.

Appendix: `occlang7` Syntax

This is the `occlang7` syntax, as found in OCCAM-LIB.Y and CRI.nl. To ease reading, later definitions are preceded by a 4-space tab, and the spaces between syntactic objects are removed.

```
program = block_definition_list
    | value_header
      block_definition_list
block_definition_list = block_definition
    | block_definition_list
      block_definition
alternation = ALT NEWLINE
                RIND alternative_vlist ROUTD
    | ALT NEWLINE
    | ALT replicator NEWLINE
        RIND alternative ROUTD
    | PRI ALT NEWLINE
        RIND alternative_vlist ROUTD
    | PRI ALT NEWLINE
    | PRI ALT replicator NEWLINE
        RIND alternative ROUTD
alternative = guarded_alternative
    | alternation
    | specification
      alternative
    | channel '?' CASE NEWLINE
        RIND variant_vlist ROUTD
    | channel '?' CASE NEWLINE
    | boolean '&' channel '?' CASE NEWLINE
        RIND variant_vlist ROUTD
    | boolean '&' channel '?' CASE NEWLINE
alternative_vlist = alternative
    | alternative_vlist
      alternative
block_definition = PROC name '(' formal_olist ')' NEWLINE
                     RIND procedure_body ROUTD
                    ':' NEWLINE
    | data_type_clist FUNCTION name '(' formal_olist ')' NEWLINE
        RIND function_body ROUTD
      ':' NEWLINE
case_input = channel '?' CASE NEWLINE
               RIND variant_vlist ROUTD
    | channel '?' CASE NEWLINE
choice = guarded_choice
    | conditional
    | specification
      choice
choice_vlist = choice
    | choice_vlist
      choice
conditional = IF NEWLINE
                RIND choice_vlist ROUTD
    | IF NEWLINE
    | IF replicator NEWLINE
        RIND choice ROUTD
construction = sequence | conditional | selection | loop
    | parallel | alternation
definition = value_definition
    | ANY FUNCTION name '(' ')' NEWLINE
        RIND function_body ROUTD
      ':' NEWLINE
    | data_type_clist FUNCTION name '(' formal_olist ')' IS expression_list ':' NEWLINE
    | specifier name RETYPES element ':' NEWLINE
    | block_definition
function_body = value_process
guarded_alternative = guard
                        RIND process ROUTD
guarded_choice = boolean NEWLINE
                   RIND process ROUTD
loop = WHILE boolean NEWLINE
         RIND process ROUTD
option = case_expression_clist NEWLINE
           RIND process ROUTD
    | ELSE NEWLINE
        RIND process ROUTD
    | specification
      option
option_vlist = option
    | option_vlist
      option
parallel = PAR NEWLINE
             RIND process_vlist ROUTD
    | PAR NEWLINE
    | PAR replicator NEWLINE
        RIND process ROUTD
    | PRI PAR NEWLINE
        RIND process_vlist ROUTD
    | PRI PAR NEWLINE
    | PRI PAR replicator NEWLINE
        RIND process ROUTD
    | placedpar
placedpar = PLACED PAR NEWLINE
              RIND placedpar_vlist ROUTD
    | PLACED PAR NEWLINE
    | PLACED PAR replicator NEWLINE
        RIND placedpar ROUTD
    | PROCESSOR expression NEWLINE
        RIND process ROUTD
placedpar_vlist = placedpar
    | placedpar_vlist
      placedpar
procedure_body = process
process = SKIP NEWLINE
    | STOP NEWLINE
    | action | construction | instance | case_input
    | specification
      process
    | allocation
      process
process_vlist = process
    | process_vlist
      process
selection = CASE selector NEWLINE
              RIND option_vlist ROUTD
    | CASE selector NEWLINE
sequence = SEQ NEWLINE
             RIND process_vlist ROUTD
    | SEQ NEWLINE
    | SEQ replicator NEWLINE
        RIND process ROUTD
specification = declaration | abbreviation | definition
valof = VALOF NEWLINE
          RIND process
          RESULT expression_list NEWLINE ROUTD
    | specification
      valof
value_process = valof
```

```
variant = tagged_list NEWLINE
            RIND process ROUTD
    | specification
      variant
variant_vlist = variant
    | variant_vlist
      variant
abbreviation = specifier name IS element ':' NEWLINE
    | name IS element ':' NEWLINE
    | value_abbreviation
action = assignment NEWLINE
    | input NEWLINE
    | output NEWLINE
allocation = PLACE name AT expression ':' NEWLINE
declaration = type name_clist ':' NEWLINE
guard = input NEWLINE
    | boolean '&' input NEWLINE
    | boolean '&' SKIP NEWLINE
instance = name '(' actual_olist ')' NEWLINE
tagged_protocol = scalar NEWLINE
    | scalar ';' sequential_protocol NEWLINE
tagged_protocol_vlist = tagged_protocol
    | tagged_protocol_vlist
      tagged_protocol
value_abbreviation = VAL specifier name IS expression ':' NEWLINE
    | VAL name IS expression ':' NEWLINE
value_definition = PROTOCOL name IS sequential_protocol ':' NEWLINE
    | PROTOCOL name NEWLINE
        RIND CASE NEWLINE
          RIND tagged_protocol_vlist ROUTD ROUTD
      ':' NEWLINE
    | PROTOCOL name NEWLINE
        RIND CASE NEWLINE ROUTD
      ':' NEWLINE
    | VAL specifier name RETYPES expression ':' NEWLINE
value_header = value_abbreviation | value_definition
    | value_header
      value_abbreviation
    | value_header
      value_definition
actual = element %dprec 2
    | expression     %dprec 1
actual_clist = actual | actual_clist ',' actual
actual_olist =  | actual_clist
assignment = variable_clist ASSIGN expression_list
base = expression
boolean =  expression
case_expression = expression
case_expression_clist = case_expression | case_expression_clist ',' case_expression
channel = element
delayed_input = channel '?' AFTER expression
expression_list = name '(' expression_olist ')' %dprec 2
    | expression_clist                          %dprec 1
formal = specifier name_clist | VAL specifier name_clist
formal_clist = formal | formal_clist ',' formal
formal_olist =  | formal_clist
indef_type = INDEF_SPEC type
    | INDEF_SPEC indef_type
    | '[' expression ']' indef_type
input = channel '?' input_item_slist
    | channel '?' CASE tagged_list
    | delayed_input
input_item = variable | variable DOUBLE_COLON variable
input_item_slist = input_item | input_item_slist ';' input_item
output = channel '!' output_item_slist      %dprec 1
    | channel '!' scalar ';' output_item_slist  %dprec 2
output_item = expression | expression DOUBLE_COLON expression
output_item_slist = output_item | output_item_slist ';' output_item
replicator =  name '=' base FOR count
selector = expression
sequential_protocol = simple_protocol_slist
simple_protocol_slist = simple_protocol | simple_protocol_slist ';' simple_protocol
specifier = type | indef_type
tagged_list = scalar | scalar ';' input_item_slist
variable = element
variable_clist = variable | variable_clist ',' variable
andor_expression = andor_expression andor_operator operand
    | operand andor_operator operand
comms_type = CHAN OF protocol
    | TIMER
    | PORT OF non_comms_type
conversion = primitive_type operand
    | primitive_type ROUND operand
    | primitive_type TRUNC operand
count = expression
element = element '[' subscript ']'
    | '[' element FROM subscript FOR count ']'
    | scalar
expression = monadic_operator operand
    | operand dyadic_operator operand
    | andor_expression
    | conversion
    | operand
    | MOSTPOS data_type
    | MOSTNEG data_type
expression_clist = expression | expression_clist ',' expression
expression_olist =  | expression_clist
operand = element | literal | table | '(' expression ')'
    | name '(' expression_olist ')'
primitive_type = comms_type
    | data_type
protocol = name | simple_protocol | ANY
simple_protocol = non_comms_type
    | counting_type DOUBLE_COLON INDEF_SPEC non_comms_type
subscript =  expression
table = table '[' subscript ']'
    | '[' expression_clist ']'
    | '[' table FROM subscript FOR count ']'
    | string
type = primitive_type
    | '[' expression ']' type
non_comms_type = data_type
    | '[' expression ']' non_comms_type
andor_operator = AND | OR
byte = BYTE_CONSTANT
counting_type = BYTE
    | INT
    | INT16
    | INT32
    | INT64
data_type = counting_type
    | BOOL
    | REAL32
    | REAL64
data_type_clist = data_type | data_type_clist ',' data_type
dyadic_operator = AFTER | BITAND | BITOR | MINUS | PLUS | REM
    | TIMES | RIGHT_OP | LEFT_OP | AND_OP | OR_OP | LE_OP | GE_OP
    | XOR_OP | NE_OP | '-' | '+' | '*' | '/' | '\\' | '<' | '>'
    | '='
integer = INT_CONSTANT | HEX_CONSTANT
literal = integer
    | byte
    | integer '(' data_type ')'
    | byte '(' data_type ')'
    | real '(' data_type ')'
    | TRUE | FALSE
monadic_operator = BITNOT | MINUS | NOT | SIZE | '~' | '-'
name = IDENTIFIER
name_clist = name | name_clist ',' name
real = REAL_CONSTANT
scalar = name
string = STRING_LITERAL | string_head STRING_LITERAL_END
string_head = STRING_LITERAL_START | string_head STRING_LITERAL_MIDDLE
```


Acknowledgments

The author would like to thank: Gavin Crate, author and maintainer of the Transputer Emulator, for his software which has been invaluable in this investigation; Kevin Chalmers, whose help applying the CSP proof tool FDR4 to the author's FIFO variants was critical in understanding Formal Verification issues; Brian Vinter and Carl-Johannes Johnsen, and the rest of the SME team at the Niels Bohr Institute, for their indispensable work in reviving and advancing the Transputer design; and finally, the reviewers, for helping me to clarify certain issues.

References

[1] INMOS Ltd. *occam 2 Reference Manual*. Prentice Hall, 1988. http://www.transputer.net/obooks/obooks.asp.

[2] INMOS Ltd. *occam2 toolset user manual - part 1*. INMOS, 1991. www.transputer.net/prog/72-tds-275-02/otdsug1.pdf.

[3] Carl-Johannes Johnsen, Kenneth Skovhede, Brian Vinter, Lindsay O'Brien Quarrie, and Lawrence J. Dickson. Implementing a Transputer for FPGA in Less Than 800 Lines of Code. In Jan Bækgaard Pedersen, Kevin Chalmers, Marc L. Smith, Kenneth Skovhede, Brian Vinter, and Peter Welch, editors, *Communicating Process Architectures 2018*. IOS Press, Amsterdam, The Netherlands, August 2018.

[4] Brian Vinter and Kenneth Skovhede. Synchronous Message Exchange for Hardware Designs. In Kevin Chalmers, Jan Bækgaard Pedersen, Frederick R. M. Barnes, Jan F. Broenink, Ruth Ivimey-Cook, Adam Sampson, and Peter Welch, editors, *Proceedings of Communicating Process Architectures 2015*. IOS Press, Amsterdam, The Netherlands, August 2015.

[5] William Harwood. *Software errors could have destroyed Boeing Starliner, NASA says*. CBS News, February 7, 2020. https://www.cbsnews.com/news/boeing-faulted-for-starliner-software-errors-that-marred-test-flight.

[6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating system principles*. SOSP '09, 2009.

[7] INMOS Ltd. *Transputer Instruction Set, a compiler writer's guide*. Prentice Hall, 1988. http://www.transputer.net/iset/iset.asp.

[8] Brian Vinter, Lawrence J. Dickson, and James Dibley. Workshop on Translating CSP-based Languages to Common Programming Languages. In Jan Bækgaard Pedersen, Kevin Chalmers, Marc L. Smith, Kenneth Skovhede, Brian Vinter, and Peter Welch, editors, *Communicating Process Architectures 2018*. IOS Press, Amsterdam, The Netherlands, August 2018. Workshop, https://github.com/SpaceSciencesCorp/Hard-Stuff-Language/CPA-Workshop-2018c.pdf.

[9] Lawrence J. Dickson and Jeremy M. R. Martin. Rigorous Timing, Static OCCAM, and Classic CSP: Formal Verification for the Internet of Things. In Jan Bækgaard Pedersen, Kevin Chalmers, Jan F. Broenink, Brian Vinter, Kevin Vella, and Peter Welch, editors, *Communicating Process Architectures 2017*. IOS Press, Amsterdam, The Netherlands, August 2017. Fringe Presentation, http://wotug.cs.unlv.edu/files/CPA2017/presentations/presentation-08.pdf.

[10] Lawrence J. Dickson and Jeremy M. R. Martin. Rigorous Timing, Static OCCAM, and Classic CSP: Mathematical Ground Truth. In Jan Bækgaard Pedersen, Kevin Chalmers, Jan F. Broenink, Brian Vinter, Kevin Vella, and Peter Welch, editors, *Communicating Process Architectures 2017*. IOS Press, Amsterdam, The Netherlands, August 2017. Fringe Presentation, http://wotug.cs.unlv.edu/files/CPA2017/presentations/presentation-24.pdf.

[11] Wikipedia. *Preorder*. Wikimedia Foundation Inc, 2020. https://en.wikipedia.org/wiki/Preorder.

[12] U. Franke, H. Fritz, A. Kuehnle, and J. Schick. Transputers on the Road. In *Transputer Applications and Systems, Proceedings of World Transputer Congress 1993*, volume 1, pages 1–17. IOS Press, Amsterdam, The Netherlands, 1993.

[13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[14] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. University of Kent at Canterbury, June 2003. http://frmb.org/pubs/fred-phd-thesis.pdf.

[15] Tor Skovsgaard, Patrick Dyhrberg Sørensen, Lawrence J. Dickson, Lindsay O'Brien Quarrie, and Brian Vinter. A Concurrent Data Collection Environment for WAsteful COMmunication SAtellite System. In Jan Bækgaard Pedersen, Kevin Chalmers, Jan F. Broenink, Brian Vinter, Kevin Vella, and Peter Welch, editors, *Communicating Process Architectures 2017*, pages 187 – 196. IOS Press, Amsterdam, The Netherlands, August 2017.

[16] A. W. Roscoe. Denotational semantics for occam. *International conference on concurrency*, Seminar on concurrency, CONCURRENCY 1984. Springer, 1984. urlhttps://link.springer.com/chapter/10.1007/3-540-15670-4_15.

[17] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. Technical monograph PRG-53, *Oxford University Computing Laboratory*, February 1986. https://www.cs.ox.ac.uk/files/3376/PRG53.pdf.

[18] M. H. Goldsmith, A. W. Roscoe, and B. G. O. Scott. Denotational semantics for occam2. Technical monograph PRG-108, *Oxford University Computing Laboratory*, June 1993. https://ora.ox.ac.uk/objects/uuid:edc1c60a-7f84-4a24-88d6-cc31a95fa366/download_file?file_format=pdf&safe_filename=47.pdf&type_of_work=Report.

[19] Lawrence J. Dickson. Supporting documentation for 'OCCAM-equivalent syntax for pure singleton descent structure'. Space Sciences Corporation, 26 February 2020. https://github.com/SpaceSciencesCorp/Hard-Stuff-Language/tree/master/occorg3.

[20] Lawrence J. Dickson. OCCAM Super-Entities and the Inline Value Process Question. Space Sciences Corporation, 31 January 2020. https://github.com/SpaceSciencesCorp/Hard-Stuff-Language/tree/master/occorg3/ANY-FUNCTION-construction.pdf.