Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

Compiling CSP

Frederick R.M. BARNES

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.
frmb@kent.ac.uk

Abstract. CSP, Hoare's Communicating Sequential Processes, is a formal language for specifying, implementing and reasoning about concurrent processes and their interactions. Existing software tools that deal with CSP directly are largely concerned with assisting formal proofs. This paper presents an alternative use for CSP, namely the compilation of CSP systems to executable code. The main motivation for this work is in providing a means to experiment with relatively large CSP systems, possibly consisting millions of concurrent processes — something that is hard to achieve with the tools currently available.

Keywords. CSP, compilers, occam-pi, concurrency

Introduction

Hoare's CSP [1,2] is a process algebra used to describe concurrent systems and their interactions. CSP allows for reasoning about a system's *denotational* semantics in addition to the relatively straightforward *operational* semantics. A significant use of CSP remains in formal reasoning, for example, with the occam2 [3] programming language [4,5], and more recently occam-π [6,7].

This paper presents another use for CSP, as a language for compilation to executable code. This differs from CSP tools such as FDR [8], that are designed for formal reasoning — e.g. proving that one system is a refinement of another. The work described here is much less complete in the formal sense; FDR will cater for all possible traces of a CSP system, the code generated by the compiler described here executes an arbitrary trace of the system. That trace may be different each time the program is executed, or the same, consistent with the expected rules of non-determinism. The advantage of the work described here is that it allows relatively large CSP systems to be exercised, since state-space explosion is not an issue.

The compiler used by this system is an experimental occam-π compiler, intended to replace the existing modified Inmos occam compiler used by KRoC [9,10]. Although the compiler is designed primarily for occam-π, its structure allows other languages and code-generation targets to be supported.

Section 1 describes the motivation for this work. An overview of the compiler infrastructure is given in section 2, followed by implementation details of the CSP support in section 3. Initial conclusions, including a brief discussion of performance, and details of future work are given in section 4.

1. Motivation

The main motivation for this work is to support the investigation and experimentation of large CSP systems, possibly containing millions of concurrent processes. The system also provides

the ability to experiment with CSP's operational semantics — e.g. introducing prioritised external choice as defined by $\mathcal{CSPP}$ [11].

Supporting the implementation also serves to exercise new multi-way synchronisation mechanisms implemented inside the occam-π run-time system, which are based on the techniques described in [12].

2. The NOCC compiler

The majority of the work described here has been implemented inside an experimental occam-π compiler named 'NOCC' (new occam-π compiler). Unlike many compilers currently available, NOCC is not centred around a specific language or code-generation target. Instead, NOCC attempts to provide an extensible compiler framework, into which new language and code-generation 'modules' can be added.

The support for CSP within the compiler grew out of an experimental occam-π extension, allowing specifications of trace patterns (in the form of CSP equations) to be attached to occam-π channel-bundle declarations. This reflects on another feature of the compiler, namely the ability to support multiple source languages in a single input file. Likewise, the compiler is capable of generating multiple output formats, e.g. for mixed hardware/software compilation.

Building compilers this way is certainly not a new idea [13], and there has been work on multi-language and multi-target compilers elsewhere. Where this compiler differs from others is in its treatment of concurrency. The occam/CSP concurrency model has been around for some time, yet there have been few compilers capable of taking full advantage of it. Creating systems consisting of millions of interacting concurrent processes places some peculiar requirements on the compiler. The two most important are efficient memory allocation (for processes at run-time) and safety checks (freedom from parallel race-hazard and aliasing errors). Also important is the efficiency of the generated code, particularly where large numbers of processes are involved — a responsibility shared with the run-time system. It appears that many existing compilers fail to meet all these demands simultaneously, though some are heading in that direction. It should be noted that these compilers, by and large, generally target sequential languages, or languages where concurrency has been added as an afterthought. Checking the safety of parallel programs in cases where the language does not rigorously police concurrency can be difficult [14] — as is the case with 'threads and locks' models of concurrency (e.g. Java threads [15,16] and POSIX threads [17]).

In terms of compiler research, NOCC does not appear to add anything substantial to what is already known about building compilers. However, it is maybe better suited to the compilation of very finely grained parallel programs, compared with other compilers. In some ways, the general structure of NOCC is similar to that of the existing Inmos occam compiler (now heavily modified for occam-π), albeit a somewhat modernised version of it.

General information pertaining to NOCC can be found at [18]. It is worth noting that NOCC is a very experimental compiler, and that the C language (in which NOCC is written) might not be the most suitable language for compiler implementation. The style of the compiler code itself is not too dissimilar to *aspect-orientation*, which has been used successfully in building other compilers [19]. The particular feature common to both is the ability to alter the behaviour of compiler parts at run-time. In aspect-orientation this is done in the language; in NOCC it is handled by changing function-pointers in structures.

2.1. Structure of NOCC

Figure 1 shows the structure of the compiler as a series of tree-passes, that transform the parse-tree at each stage. Modules inserted into the compiler may add their own passes, if the

existing structure of the compiler is incapable of supporting something required.

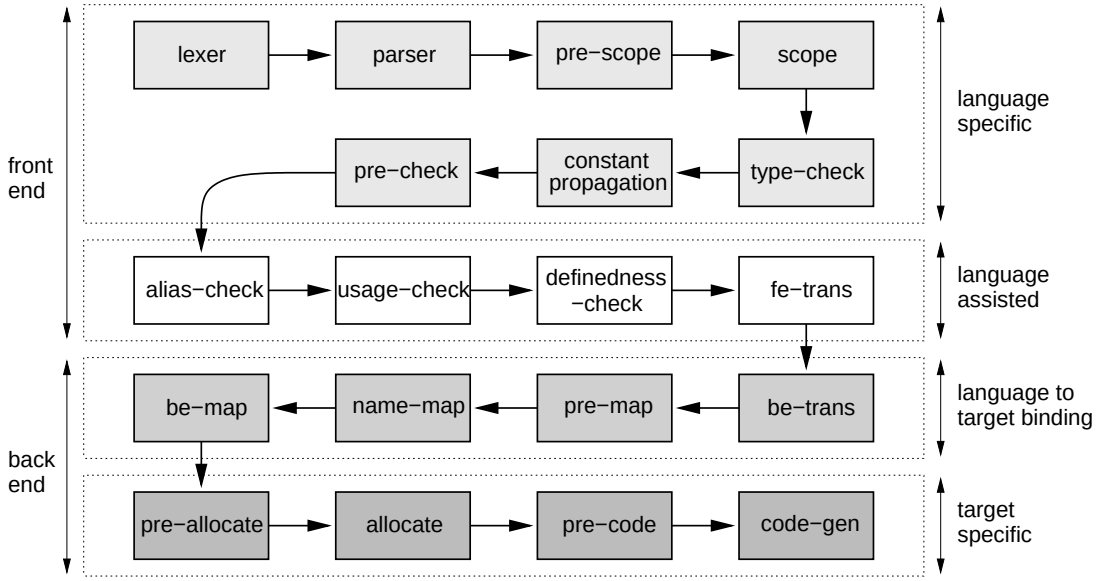

Figure 1. Structure of the NOCC compiler

Most of the compiler passes are obviously named. The ‘pre-scope’ pass is used to perform tree rewriting prior to scoping, e.g. for pushing name-nodes up the tree, where those names are declared inside their effective scope. The ‘pre-check’ pass runs prior to the parallel-usage, aliasing and definedness checks, and is used to set up tree nodes that require special checking. The ‘fe-trans’ and ‘be-trans’ passes successively simplify the parse tree, making it more suitable for code-generation. The ‘be-trans’ pass is the first point at which target information is available. The ‘name-map’ pass generates nodes representing memory reservations and memory references — e.g. describing the memory requirements of an occam-π ‘`INT`’ variable on a 32-bit machine. The ‘be-map’ pass is primarily concerned with expression evaluation, handling the allocation of temporary variables where required. The ‘pre-allocate’ and ‘allocate’ passes perform memory allocation proper. For the default *virtual transputer* ETC [20] target, this includes allocations into *workspace* (local process stack), *vectorspace* (large process local objects) and *mobilespace* (a global memory space). These are statically allocated memories, independent of memory dynamically allocated and managed at run-time.

The only code-generator currently supported by the compiler generates an extended ETC code. This is transformed into native code by the existing KRoC tool-chain. Another code-generator is under development that attempts to generalise all register-based architectures (which would ultimately remove the need for the existing native-code translator). The advantage here is to avoid inefficiencies introduced by the intermediate code.

2.2. Parse Tree Structures

Internally, the representation of parse trees is generalised. This greatly aids in the automatic processing of tree-nodes — e.g. when constructing new nodes in the parser, or when rewriting trees as part of compiler transformations. It also allows the set of node types to be modified at run-time, e.g. to support compiler extensions that add their own node types. More importantly, it enables code in one part of the compiler to operate on another part’s tree nodes without explicit knowledge of them. This is substantially different from the existing Inmos occam compiler, where the set of tree nodes is fixed, and the code makes extensive use of ‘`switch()`’ statements. In terms of data structures, what NOCC ends up with is not entirely unlike inheritance in *object orientation*, except that the equivalent “class hierarchies”

are setup at run-time. The similarity with *aspect orientation* is the ability to change these at run-time.

Each tree node in the compiler is associated with a particular *node-tag*. These provide a way of identifying the different tree nodes, e.g. "`MCSPSEQ`" and "`MCSPPAR`". Associated with each node-tag is a *node-type*, usually applied to a set of related tags. For example, '`MCSPSEQ`' and '`MCSPPAR`' are tags of a "`mcsp:dopnode`" type (dyadic operator). It is this node-type that defines the operations performed on a node in different compiler passes — by having the node type structure contain called function pointers. A compromise between speed and size is made here — the cost of calling a function is mostly constant, but the code called (for a particular node type) will often have to perform a series of if-else tests to determine which node tag is involved (and therefore how the pass should proceed). A constant-cost '`switch()`' statement cannot be used as node tags are allocated dynamically.

Figure 2 shows the C structures used to represent parse-tree nodes, together with their linkage to node-tags and node-types.

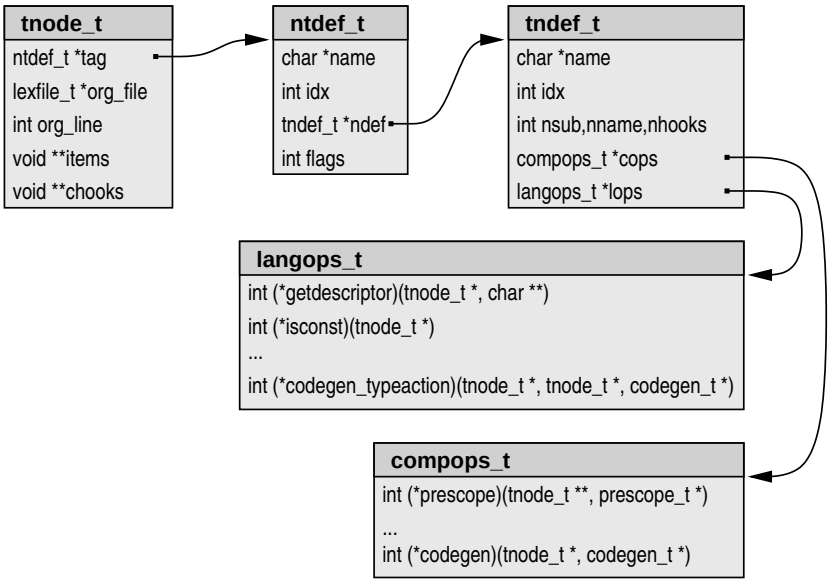

Figure 2. NOCC parse-tree structures

A lot of the node-specific code that sits behind these function pointers is concerned with parse-tree rewriting. To aid these, a tree-rewriting mini-language is being developed within the compiler. By generalising this it becomes possible for source codes to define their own language extensions, together with the tree transformations required to produce compilable parse-trees. In some ways, this is heading towards a compiler that implements a basic feature set, then loads the required language 'source' at run-time before compiling code proper; not entirely unlike compilers such as Peri Hankey's "language machine" [21].

2.3. Deficiencies

With regard to NOCC's plug-in extensions, there are currently no guarantees (beyond rigorous testing) that two or more extensions will coexist sensibly — even then, testing at compiler-development time may not be an option; e.g. new syntactic structures defined inside source files to be compiled. This issue will be addressed in the future once the compiler supports enough of the occam-π language to be useful in practical applications[1]; existing work has already investigated this issue [22], but for a compiler written in Scheme. The poten-

[1] At the time of writing, simple programs compile successfully but without many of the checks that the existing compiler applies. Basic tests for channel and variable parallel-usage have been implemented successfully.

tial problems lie not so much with incompatibilities in tree nodes, but with incompatibilities in the language grammar — e.g. interaction between a module that provides multi-variable declarations, "`INT a,b:`", and one that provides for initialising declarations, "`INT x=42:`". The resulting problem is whether "`INT f=3, g=24:`" or "`INT f,g = 3,24:`" would parse, and if so, whether either would have the expected semantics (multiple initialised integer variables).

3. Compiling CSP

This section describes the details of generating executable code from CSP expressions, within the NOCC compiler. It also sheds some further light on the details of programming within NOCC. The resulting language supported by the compiler is loosely termed 'MCSP' (machine-readable CSP), but is currently incompatible with other machine-readable CSP representations (e.g. that used by FDR).

3.1. Syntax

The way in which NOCC operates requires front-end language modules to supply the parser grammar BNF-style or as textual DFAs (deterministic finite automaton, or state machines). This allows the language syntax to be readily changed, particularly within plug-in modules which modify the grammar — or even within source files themselves. The following, for example, describes how CSP fixpoints are represented (and is the literal text fed to the compiler at run-time):

```
mcsp:fixpoint ::= [ 0 @@@ 1 ] [ 1 mcsp:name 2 ] [ 2 @@. 3 ]
                  [ 3 mcsp:process 4 ] [ 4 {<mcsp:fixreduce>} -* ]
```

This is a DFA representation that will parse "`@`" followed by a name, "`.`" and a process. '`mcsp:fixreduce`' is a reduction rule that specifies how to generate the tree-node result. In this particular case it is a *generic reduction* rule — a program for a miniature stack-machine that operates on parse-tree nodes:

```
parser_register_grule ("mcsp:fixreduce",
    parser_decode_grule ("SN0N+N+VC2R-", mcsp.tag_FIXPOINT));
```

The '`tag_FIXPOINT`' is a reference to a *node-tag*, defined earlier with:

```
mcsp.tag_FIXPOINT = tnode_newnodetag ("MCSPFIXPOINT", NULL,
    mcsp.node_SCOPENODE, NTF_NONE);
```

This further references the '`node_SCOPENODE`' *node-type*, the compiler structure onto which the MCSP module attaches various functions to handle transformations in the different compiler passes. At the parser top-level, the '`mcsp:fixpoint`' rule is incorporated statically in the definition of '`mcsp:process`'. However, it could be added dynamically with the BNF-style rule:

```
mcsp:process +:= mcsp:fixpoint
```

Using these descriptions, the resulting parser is able to handle fixpoints in the input such as:

```
PROCESS (e) ::= @x.(e -> x)
```

This represents a process that continually engages on the parameterised event '*e*', and could form a whole MCSP program. Figure 3 shows the resulting parse-tree structure, prior to scoping.

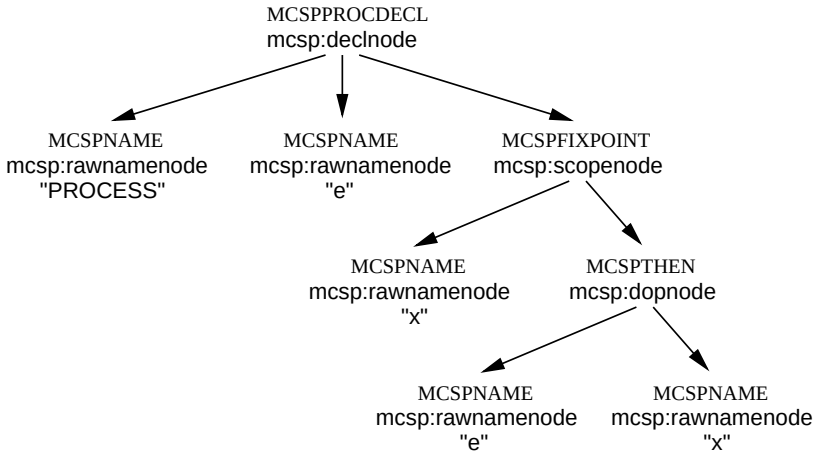

Figure 3. Tree structure after parsing

3.2. Semantics

The semantics implemented by the compiler remain faithful to CSP. External choice between two or more offered events is resolved *arbitrarily*. This allows, for example, the following program to choose '*e*' over '*f*' (if both are ready):

```
PROCESS (e,f) ::= @x.((e -> x) [] (f -> x))
```

There is a difference concerning the scoping of names. In the short examples seen so far, all events are bound to parameters. This need not be the case, however. Unbound events are automatically inserted into parameter lists by the compiler and collected together at the top-level. In traditional CSP, which does not support parameterised events, the definition of something (e.g. 'PROCESS') is entirely in the meta-language — the right-hand-sides of such definitions could be literally substituted without changing the semantics of the system. This is not the case here, for example:

```
FOO ::= e -> SKIP
BAR (e) ::= e -> FOO
```

There are two '*e*' events in this system — one free and one bound to 'BAR'. It is equivalent to the system:

```
BAR (f) ::= f -> e -> SKIP
```

Literal substitution of 'FOO' would produce a different system.

The two behaviours not currently supported are interleaving and interrupts. Interrupts are not too much of a concern for implementation — event synchronisations in affected processes becomes a choice between the existing event and the interrupt event, followed by the equivalent of a 'goto' into the interrupt process.

3.2.1. Interleaving

Interleaving gives rise to difficulties in the implementation, but only where 'shared' events are involved. For example:

```
PROCESS (e,f,g) ::= (e -> f -> SKIP) ||| (e -> g -> SKIP)
```

In such a system, either the left-hand-side synchronises on '*e*' or the right-hand-side does, but they do not synchronise on '*e*' between themselves. The implementation of events is handled using a new version of the multi-way synchronisation described in [12]. At present,

however, these only support synchronisation between *all* processes enrolled on the event — providing an implementation for interleaving is complex.

A possible solution is to re-write the affected expressions, separating out the affected events. This, however, does not scale well — memory occupancy and run-time cost increase.

The solution to be implemented involves a change to the implementation of multi-way events, such that interleaving is explicitly supported. Parallel synchronisation is a *wait-for-all* mechanism; interleaving is *wait-for-one*. These represent special cases of a more general synchronisation mechanism, *wait-for-n*. Although CSP provides no model for this (the denotational semantics would be very complex), it does have uses in other applications [23], and is something that could be supported for occam-π.

3.3. Code generation

Supporting multiple back-end targets creates some interesting challenges for the compiler. Currently this is handled by abstracting out 'blocks' and 'names', that describe the run-time memory requirements. The compiler uses this information to allocate variables, parameters and other memory-requiring structures.

Figure 4 shows the transformed sub-tree for the '`FIXPOINT`' node from figure 3, simplified slightly. The new nodes whose names start 'krocetc' belong to the KRoC ETC code-generator within the compiler. The CSP (or occam-π) specific portions of the compiler are only aware of these nodes as back-end blocks, block-references, names or name-references.

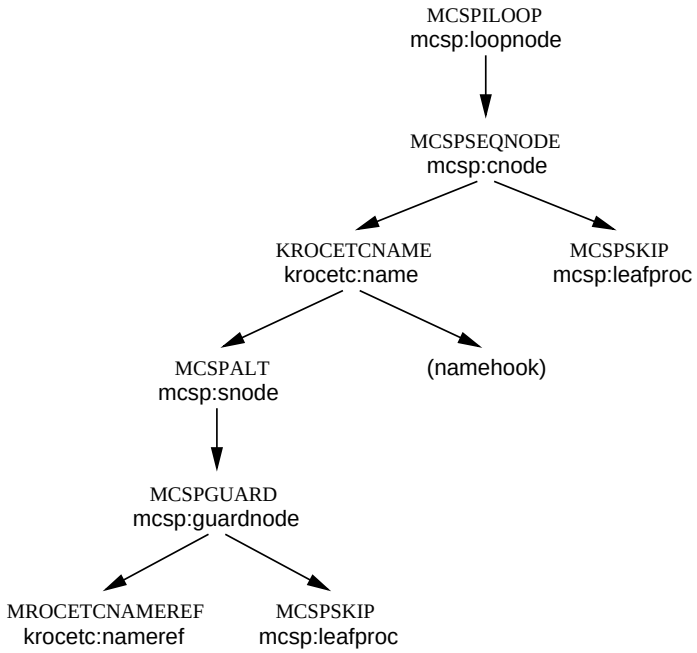

Figure 4. Sub-tree structure after name-map

The '`krocetc:nameref`' node is a reference to the parameterised event involved, which has since been transformed into a '`krocetc:name`' describing memory requirements and levels of indirection. The '`krocetc:name`' node left in the tree reserves space for the occam-π style '`ALT`' (needed by the run-time). Handling these in a language and target independent manner is only partially complete. The back-end definition within the compiler maintains details of the memory required for constructs such as '`ALT`'s and '`PAR`'s, and this is a fixed set. In the future it may be desirable to generalise these features, but so far this has not been necessary. The combination of occam-π and ETC appears to produce a fairly broad set of compiler features, certainly more than sufficient for purely sequential languages, or for languages with more primitive concurrency models (e.g. threads and locks).

The complete resulting parse-tree is much larger than the ones shown here. Inserting the various back-end nodes will often double the size of the existing tree, including name-nodes for entities such as a procedure's return-address, or static-links to higher lexical levels. Furthermore, the top-level process (and main program) generated by the compiler is not the program's top-level process. It is instead the parallel composition of the program's top-level process with a newly created 'environment' process. It is this environment process which produces the program's output at run-time, by writing bytes to the KRoC top-level 'screen' channel. For example, the earlier system:

```
PROCESS (e) ::= @x.(e -> x)
```

is transformed by the compiler into the following (illegal) code:

```
PROCESS (e) ::= @x.(e -> x)
ENVIRONMENT (out,e) ::= @z.(e -> out!"e\n" -> z)
SYSTEM (screen) ::= (PROCESS (k) || ENVIRONMENT (screen,k)) \ {k}
```

When executed, this system simply outputs a continuous stream of "e"s, each on a new line. Normally such code would be illegal since the '`out`' event is really an occam channel; the compiler also currently lacks the support to parse the output syntax shown. An explicit syntax and semantics for handling occam-style communication may be added at a later date.

The ETC generated by the compiler is not too dissimilar to what might be expected from a comparable occam-π program. The main difference is the use of new multi-way synchronisation instructions, implemented in the run-time kernel (a modified CCSP [24]). Because the CSP parts of the compiler generate code intended for the KRoC run-time, there is no reason why the two cannot be used together within the same program — once the necessary support for multi-way synchronisation has been incorporated into the occam-π code within NOCC.

3.3.1. Other compiler output

In addition to the ETC or other code output, the compiler generates a '`.xlo`' file. This is an XML file that describes the compiled output. These files are read back by the compiler when separately compiled code is incorporated, e.g. through a '`#USE`' directive. Included in the output is all the information necessary to instance the separately compiled code, typically memory requirements and entry-point names.

Optionally included is a digitally signed hash-code of the generated code and related information in the 'xlo' file. Public/private key-pairs are used, allowing verification that code was generated by a particular compiler. This will typically be useful in distributed occam-π mobile-agent systems, when a node that receives precompiled code needs to guarantee the origin of that code.

3.4. Run-time support

In order to support MCSP programs, multi-way synchronisation instructions have been added to the KRoC run-time. These consist of modified '`ALT`' start, end and wait instructions plus additional enable-guard and disable-guard instructions. As can be seen in figure 4, single synchronisations are transformed into an '`ALT`' structure with only one guard — an explicit synchronisation instruction may be added in the future.

In order to support the new compiler, various other changes have been made to the translator and run-time. The most significant is the removal of the fixed *workspace-pointer* adjustment on subroutine calls and returns (the '`CALL`' and '`RET`' transputer instructions [25]). As a result, the '`CALL`' instruction no longer expects to have subroutine parameters on the stack — which would normally be copied into space reserved by the fixed workspace adjust-

ment. The new compiler instead generates 'CALL' and 'RET' instructions with an extra 'off-set' operand, that specifies the necessary workspace adjustment. This makes code-generation and the general handling of subroutine calls simpler, and potentially less expensive.

3.5. Supported constructs

Table 1 provides an overview of the supported constructs and their MCSP syntax. Although much of traditional CSP is supported, the language lacks features that may be useful from a pragmatic perspective — for example, variables and basic arithmetic. Such features move the system away from traditional CSP and more towards process calculi such as Circus [26]. Additional features currently being considered are covered in section 4.

	CSP	**MCSP**
skip	*SKIP*	`SKIP`
stop	*STOP*	`STOP`
chaos	*CHAOS*	`CHAOS`
divergence	div	`DIV`
event prefix	$e \rightarrow P$	`e -> P`
internal choice	$(x \rightarrow P) \sqcap (y \rightarrow Q)$	`(x -> P) |~| (y -> Q)`
external choice	$(x \rightarrow P) \Box (y \rightarrow Q)$	`(x -> P) [] (y -> Q)`
sequence	$P \fatsemi Q$	`P; Q`
parallel	$P \parallel Q$	`P || Q`
interleaving	$P \mid\mid\mid Q$	`P ||| Q`
hiding	$P \setminus \{a\}$	`P \ {a}`
fixpoint	$\mu X.P$	`@X.P`

Table 1. Supported MCSP constructs and syntax

A restriction is currently placed on the use of the fixpoint operator — it may only be used to specify sequential tail-call recursion, as shown in the previous examples. Effectively, anything that can be transformed into a looping structure. More general recursion using the fixpoint operator will be added at a later stage.

4. Conclusions and future work

This paper has described the basic mechanisms by which CSP programs are translated into executable code. Although the work is at an early stage and there is much more to do, it does show promise — particularly for expressibility and performance. In addition to providing a means to compile and execute CSP programs, this work serves as a good exercise for the new occam-π compiler. Importantly, the incremental addition of MCSP language features to the compiler has remained relatively simple. This is not the case for the existing occam-π compiler, where certain optimisations made early on in the compiler's development[2] make adding extensions awkward. For instance, modifying the source language syntax is trivial, as NOCC generates the actual parser from high-level BNF or DFA definitions.

4.1. Performance

A standard benchmark for occam-π is 'commstime', a cycle of processes that continuously communicate. This has been re-engineered into MCSP, where the processes involved simply

[2]The original occam compiler was designed to run in 2 megabytes of memory, which led to a fairly complex optimised implementation.

synchronise, rather than communicate data. To make benchmarking slightly simpler, the proposed sequential-replication addition (section 4.2.2) has already been added. The commstime benchmark, with a sequential 'delta', is currently implemented as:

```
PREFIX (in,out)      ::= out -> @x.(in -> out -> x)
SUCC (in,out)        ::= @x.(in -> out -> x)
DELTA (in,out1,out2) ::= @x.(in -> out1 -> out2 -> x)
CONSUME (in,report)  ::= @x.((;[i=1,1000000] in); report -> x)

SYSTEM (report)      ::= ((PREFIX (a, b) || DELTA (b, c, d)) ||
                          (SUCC (c, a) || CONSUME (d, report))) \ {a,b,c,d}
```

When compiled and executed, the program will print '`report`' on the screen for every million cycles of the benchmark. Measuring the time between printed 'report's gives a fairly accurate value for synchronisation time. Each cycle of the benchmark requires 4 complete multi-way synchronisations, or 8 individual multi-way '`ALT`'s. On a 2.4 GHz Pentium-4, the time for a complete multi-way synchronisation is approximately 170 nanoseconds (with each synchronisation involving 2 processes).

In fact, this figure is a slight over-estimation — it includes a small fraction of the cost of printing '`report`', and the loop overheads. More accurate benchmarking will require the ability to handle timers, something which is considered in the following section. When using a parallel delta, the synchronisation cost is increased to around 250 nanoseconds, giving a process startup-shutdown cost of around 160 nanoseconds.

4.2. Planned additions

This section describes some of the additions planned for the MCSP language, to be implemented in the NOCC compiler. These are largely concerned with practical additions to the language, rather than semantic updates. It should be noted that some of the planned additions potentially modify the operational semantics. However, they should not modify the denotational semantics — i.e. systems should not behave in an unexpected way.

4.2.1. Alphabetised parallel

The current parallel operator '`||`' builds its alphabet from the intersection of events on either side. This makes it difficult to express two parallel processes that synchronise on some events, but interleave on others. For example:

```
SYSTEM ::= (a -> b -> c -> SKIP) |{a,c}| (a -> b -> c -> SKIP)
```

4.2.2. Replicated processes

Table 2 shows the proposed operators to support replicated processes. In each case, the replicator may be given as '`[i=n,m]`' for replication of $(m-n)+1$ processes, or more simply as '`[n]`' for n replications. For the former, the name '`i`' is a read-only integer variable, in scope for the replicated process only.

	CSP	**MCSP**
sequential replication		`;[i=1,n] P`
parallel replication	$\|\|_{\{i=1..n\}} P$	`||[i=1,n] P`
interleave replication	$\|\|\|_{\{i=1..n\}} P$	`|||[i=1,n] P`

Table 2. Proposed replicator constructs

4.2.3. Variables and expressions

Including variables and expressions within the language adds an amount of computational power. The three types initially proposed are ‘bool’s, ‘int’s (signed 32-bit integers) and ‘string’s. These should provide enough computational functionality for many MCSP programs, at least those which we currently have in mind.

Table 3 shows the proposed additions. These, however, are not consistent with similar CSP functionality described in [2], where input and output of integers is modelled with a set of events — one for each distinct integer. Here there is only a single event and data is transferred between processes. A restriction must be imposed on outputting processes, such that outputs never synchronise with each other — i.e. one output process synchronising with one or more input processes. Interleaving outputs would be permitted, however.

	MCSP
variable declaration and scope	`name:type P`
assignment process	`name := expression`
input process	`e?name`
output process	`e!expression`
choice	`if condition then P else Q`

Table 3. Proposed variable and expression constructs

For expressions and conditions, we intend to support a ‘standard’ range of operators. For example, addition, subtraction, multiply, divide and remainder for integer expressions; equal, less-than, greater-than or equal and their inverses for integer comparisons. For strings, concatenation and equality tests only. Within the top-level ‘environment’ process, events that are used for communication will input data from the system and display it on the screen.

One further consideration, particularly for benchmarking, is a handling of time. Keeping some consistency with occam-π, a possible implementation would be a global ‘timer’ event on which all processes interleave. Input will produce the current time (in microseconds), output will block the process until the specified time is reached.

The combination of features considered here should make it possible to write reasonable extensive and entertaining CSP programs — e.g. a visualisation of the classic “dining philosophers” problem. Over time it is likely that we will want to add, modify or entirely remove features from the MCSP language. The NOCC compiler framework should make this a relatively straightforward task, but only time will tell.

References

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-13-153271-5.

[2] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[3] Inmos Limited. *occam2 Reference Manual*. Prentice Hall, 1988. ISBN: 0-13-629312-3.

[4] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational Semantics for occam2, Part 1. In *Transputer Communications*, volume 1 (2), pages 65–91. Wiley and Sons Ltd., UK, November 1993.

[5] M.H. Goldsmith, A.W. Roscoe, and B.G.O. Scott. Denotational Semantics for occam2, Part 2. In *Transputer Communications*, volume 2 (1), pages 25–67. Wiley and Sons Ltd., UK, March 1994.

[6] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent, June 2003.

[7] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In A.E. Abdallah, C.B. Jones, and J.W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[8] Formal Systems (Europe) Ltd., 3, Alfred Street, Oxford. OX1 4EH, UK. *FDR2 User Manual*, May 2000.

[9] P.H. Welch and D.C. Wood. The Kent Retargetable occam Compiler. In Brian O'Neill, editor, *Parallel Processing Developments, Proceedings of WoTUG 19*, volume 47 of *Concurrent Systems Engineering*, pages 143–166, Amsterdam, The Netherlands, March 1996. World occam and Transputer User Group, IOS Press. ISBN: 90-5199-261-0.

[10] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

[11] A.E. Lawrence. CSPP and Event Priority. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 67–92, Amsterdam, The Netherlands, September 2001. WoTUG, IOS Press. ISBN: 1-58603-202-X.

[12] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating Complex Systems. In *Proceedings of ICECCS-2006*, September 2006.

[13] Uwe Schmidt and Reinhard Völler. A Multi-Language Compiler System with Automatically Generated Codegenerators. *ACM SIGPLAN Notices*, 19(6), June 1984.

[14] Zehra Sura and Xing Fang and Chi-Leung Wong and Samuel P. Midkiff and Jaejin Lee and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 2–13, New York, NY, USA, 2005. ACM Press.

[15] B. Joy, J. Gosling, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. ISBN: 0-20-163451-1.

[16] D. Lea. *Concurrent Programming in Java (Second Edition): Design Principles and Patterns*. The Java Series. Addison-Wesley, 1999. section 4.5.

[17] International Standards Organization, IEEE. Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language], 1996. ISO/IEC 9945-1:1996 (E) IEEE Std. 1003.1-1996 (Incorporating ANSI/IEEE Stds. 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995).

[18] F.R.M. Barnes. NOCC: A New occam-pi Compiler. URL: `http://www.cs.kent.ac.uk/projects/ofa/nocc/`.

[19] Xiaoqing Wu and Barrett R. Bryant and Jeff Gray and Suman Roychoudhury and Marjan Mernik. Separation of concerns in compiler development using aspect-orientation. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1585–1590, New York, NY, USA, 2006. ACM Press.

[20] M.D. Poole. Extended Transputer Code - a Target-Independent Representation of Parallel Programs. In P.H. Welch and A.W.P. Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 187–198, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9.

[21] Peri Hankey. The Language Machine. URL: `http://languagemachine.sourceforge.net/`.

[22] Matthew Flatt. Composable and compilable macros:: you want it when? In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 72–83, New York, NY, USA, 2002. ACM Press.

[23] Mordechai Ben-Ari. How to solve the Santa Claus problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.

[24] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X.

[25] Inmos Limited. *The T9000 Transputer Instruction Set Manual*. SGS-Thompson Microelectronics, 1993. Document number: 72 TRN 240 01.

[26] J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.