Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

gCSP occam Code Generation for RMoX†

Marcel A. GROOTHUIS, Geert K. LIET, Jan F. BROENINK
Twente Embedded Systems Initiative,
Drebbel Institute for Mechatronics and Control Engineering,
Faculty of EE-Math-CS, University of Twente,
P.O.Box 217, 7500 AE, Enschede, the Netherlands
m.a.groothuis@utwente.nl

Abstract. gCSP is a graphical tool for creating and editing CSP diagrams. gCSP is used in our labs to generate the embedded software framework for our control systems. As a further extension to our gCSP tool, an occam code generator has been constructed. Generating occam from CSP diagrams gives opportunities to use the Raw-Metal occam eXperiment (RMoX) as a minimal operating system on the embedded control PCs in our mechatronics laboratory. In addition, all processors supported by KRoC can be reached from our graphical CSP tool. The *commstime* benchmark is used to show the trajectory for gCSP code generation for the RMoX operating system. The result is a simple means for using RMoX in our laboratory for our control systems. We want to use RMoX for future research on distributed control and for performance comparisons between a minimal operating system and our CTC++/RT-linux systems.

Keywords. CSP, Embedded Control Systems, Real-time, occam

Introduction

For broad acceptance of an engineering paradigm, a graphical notation and a supporting design tool is needed. This is especially the case for CSP-based software, since it is *concurrent* software. Designers draw often a kind of block diagram to indicate the flow of data along the channels that connect the processes [1-6]. Besides standardization, a graphical tool supporting the gCSP graphical notation allows for proper consistency between the diagrams and resulting concurrent code. This opens the way towards a model-driven development environment, where the diagram of the structure of the concurrent processes *is* the specification of it. The consistency between diagram and concurrent software is thus intrinsically guaranteed.

From *one* model, different kinds of code can be generated, using multiple code generators, that all use the same input data from the graphical tool. Conclusions drawn from tests on one kind of generated code can be used for another kind of generated code (for example the results of model checks with FDR2 using the CSPm generator can be applied to the executable code generated with the CTC++ generator).

CSP and formal checking are used in our labs to generate high quality control software free of deadlocks and divergence. Following our experience with the gCSP tool [7] an obvious extension is an occam code generator. gCSP already supports CSPm and (CT)C++

† This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

code generators. The generated occam code can then be used as source code for the light weight RMoX operating system [2] running on PCs. Since embedded PCs are often used to control mechatronic setups, using RMoX extended with control software will result in a small embedded control system. Section 1 gives information on gCSP and the RMoX operating system. In Section 2, the architecture and software construction issues of this gCSP occam code generation extension are treated. Section 3 presents a case study in which the *commstime* benchmark is used to test the code generation for RMoX. Section 4 concludes this paper with conclusions and our future work on gCSP/ RMoX.

1 gCSP and RMoX

1.1 gCSP

gCSP is a graphical tool for creating and editing CSP diagrams. It is based on the Graphical Modeling Language developed by Hilderink [8, 9]. CSP diagrams are dataflow diagrams, connecting processes with channels. Besides the dataflow, the concurrency structure is also indicated. The nodes in the graph are connected by two kinds of edges, namely the channels and compositional relations (see **Figure 4** for an example).

The structure of the processes including their concurrency relation is presented as a tree. This is another view of the structure in the diagram, with focus on the composition, since occam -like programs are always shaped as a strict tree-like hierarchy of SEQ, (PRI)PAR and (PRI)ALT constructs as branches and user-defined processes as leaves.

Besides editing, gCSP does basic consistency checks and can generate code from the diagrams, thus intrinsically guaranteeing consistency between diagram and resulting code. The code generation outputs of gCSP are currently CSPm, as input for model checkers like FDR2, and CTC++, the CSP / C++ library of our group. The third code generation output, namely occam, is one of the subjects of this paper.

1.2 RMoX

The RMoX operating system [2] is a small CSP based operating-system. The core is built around a stripped down version of the Linux kernel, for the low-level operating system operations, and the RMoX kernel. This RMoX kernel is an occam program that can also run within a standard Linux environment (*User Mode* RMoX). All RMoX components, like device drivers, consoles and the occam demo applications are included in this single program as occam processes.

Since RMoX is a minimal operating system based on CSP, occam and Linux, it is an interesting target OS for embedded control PCs which are often equipped with small flash based disks and a small amount of memory. Embedded control PCs do not need all functionality offered by general purpose operating systems like Linux, instead they require accurate timing for a hard real-time control loop.

The Linux basis gives the flexibility of adding existing Linux drivers for our I/O hardware to RMoX. The RMoX kernel gives us high speed CSP concurrency. The destination target platforms that can be reached by RMoX are restricted by two factors: the KRoC compiler should support it and a Linux kernel port should exist for this platform. RMoX is currently designed for Pentium based systems. Supporting smaller targets like DSPs will require much porting efforts for both KRoC and the Linux kernel.

2 The gCSP occam Code Generator

The essential architectural choice here is that the different code generators start from the same data model (i.e. data structure in the graphical editor), and that all other transformations are common to all the code generation output, see **Figure 1**.

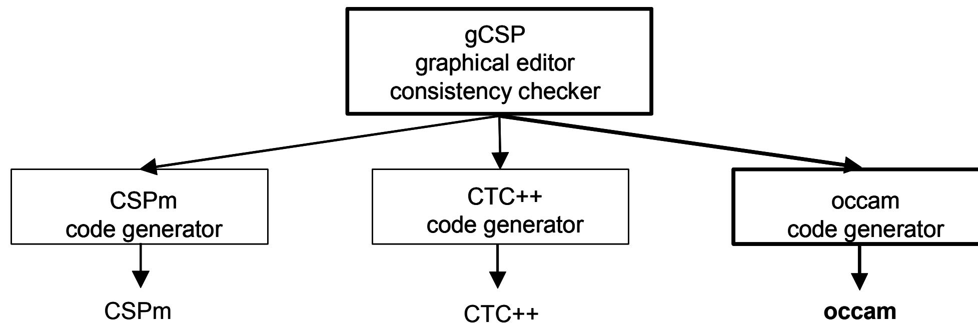

Figure 1: Global structure of gCSP with its code outputs

The occam code generator is another output next to the two existing code generator outputs gCSP already has (namely CSPm and CTC++). Since all three code generation target languages are based on CSP, it is rather obvious that the occam code generator is comparable to the other two code generators. However, there are differences which are caused by the differences in target languages that gCSP generates code for. In the current implementation, the following six differences were recognized:

1. Initialization and body of a process are not separated in occam, in contrast to CTC++.
2. At an ALT construct, the readers in the guard and the guarded process need special attention for occam code generation.
3. occam only supports input guards in an ALT construct.
4. gCSP uses different names for arithmetic data types, double and float, which are cast to `REAL64` and `REAL32` in occam respectively.
5. occam has more types than gCSP currently supports: for instance the `TIMER` type.
6. Occam uses channels instead of functions for screen output and keyboard input.

Item one implies that the code for sub processes has to be generated in-line instead of as separate functions. By using an appropriate folding editor and sophisticated comment lines, the overview of the generated occam code can be supported. However, inspecting the generated occam code should hardly be necessary. The 'real' source code is the gCSP diagram, including its code blocks to specify the algorithmic bodies of the processes.

The implementation of the ALT construct in CTC++ combines the readers in the guards with the readers in the alternative processes, thus preventing a double read action. This behaviour is different in occam. Furthermore, occam only supports input guards, whilst CTC+ also supports output guards in an ALT construct.

The sixth item implies that the generated occam code for processes that use screen output or keyboard input needs more channels than the corresponding CTC++ code. This is solved by adding 'hidden' screen or keyboard channels to an occam process if needed. These channels exist in the generated occam code, but are invisible in the gCSP model to maintain the overview. Another solution would be the use of external (linkdriver) channels to access the screen and the keyboard, but is currently not possible in gCSP to draw any-to-one and one-to-any channels.

gCSP produces the code of **Listing 1** from the producer – consumer example, as shown in **Figure 2**. The producer contains a SEQ of a code block and a writer. The consumer contains a SEQ of a reader and a code block. The producer produces data and writes it to a channel, while the consumer reads the data and writes it to the screen. The producer has a higher priority than the consumer has (the arrow above the || points to the process with the highest priority).

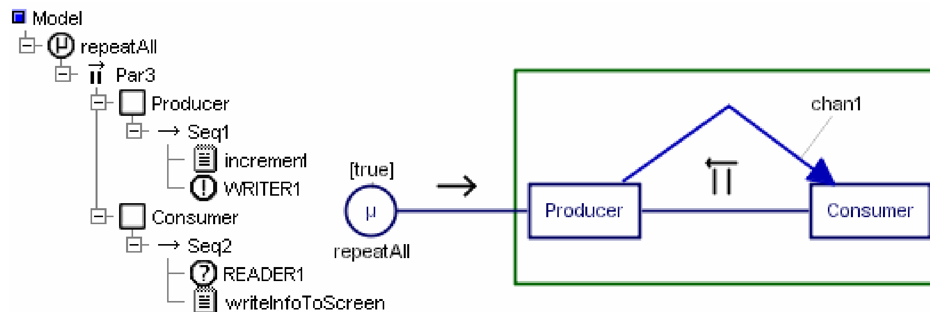

Figure 2: Producer - Consumer example in gCSP: composition tree (left) and block diagram (right).

Because these processes are small, a new *inline generation* feature has been added to the gCSP code generation to optimize the generated occam code. With this option enabled, the contents of the producer and consumer process will be generated as part of the model process instead of separate processes (compare **Listing 1** with **Listing 2**). This feature is not yet available for CTC++.

```
PROC gCSPModel(CHAN BYTE screen)
  ---Initialization
  INITIAL REAL64 y IS 0.0 :
  INITIAL REAL64 x IS 0.0 :
  CHAN REAL64 chan1:
  ---Process Body
  WHILE TRUE
    PRI PAR
      ---Producer
      SEQ
        y := y + 1.0
        IF
          y > 10.0
            y := 0.0
          TRUE
            SKIP
        chan1 ! y
      ---Consumer
      SEQ
        chan1 ? x
        out.string("Value read from channel: *n",0,screen)
        out.real64(x,0,0,screen)
        out.string("*n",0,screen)
:
```

Listing 1: occam code of Figure 2 generated by gCSP using inline generation

The gCSP generated occam code always contains one parent process with the name `gCSPModel`. This process will contain all occam code for all processes, either inline generated or using subprocesses.

3 Example: *commstime* Benchmark

To test the occam code generation in combination with RMoX, the occam *commstime* benchmark program delivered with KRoC was used as an example. **Figure 3** shows the route from a gCSP model to a running example under RMoX.

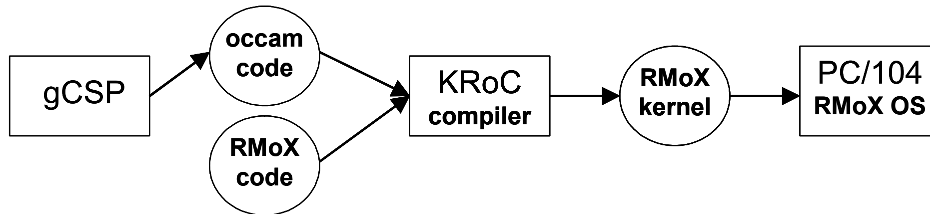

Figure 3: Overview for the gCSP to RMoX route

The first step is drawing the *commstime* example in gCSP. This results in the gCSP diagram for the *commstime* demo shown in **Figure 4**. The top right part of this figure shows four processes in parallel of which three processes send data in a circle and the fourth one, the `TimeAnalyser`, measures the loop time and the time required for the context switching. The left part of the figure shows the composition and at the bottom right part, the contents of the first process in the three, the successor, is shown. The internals of the other processes are not shown, but they are comparable.

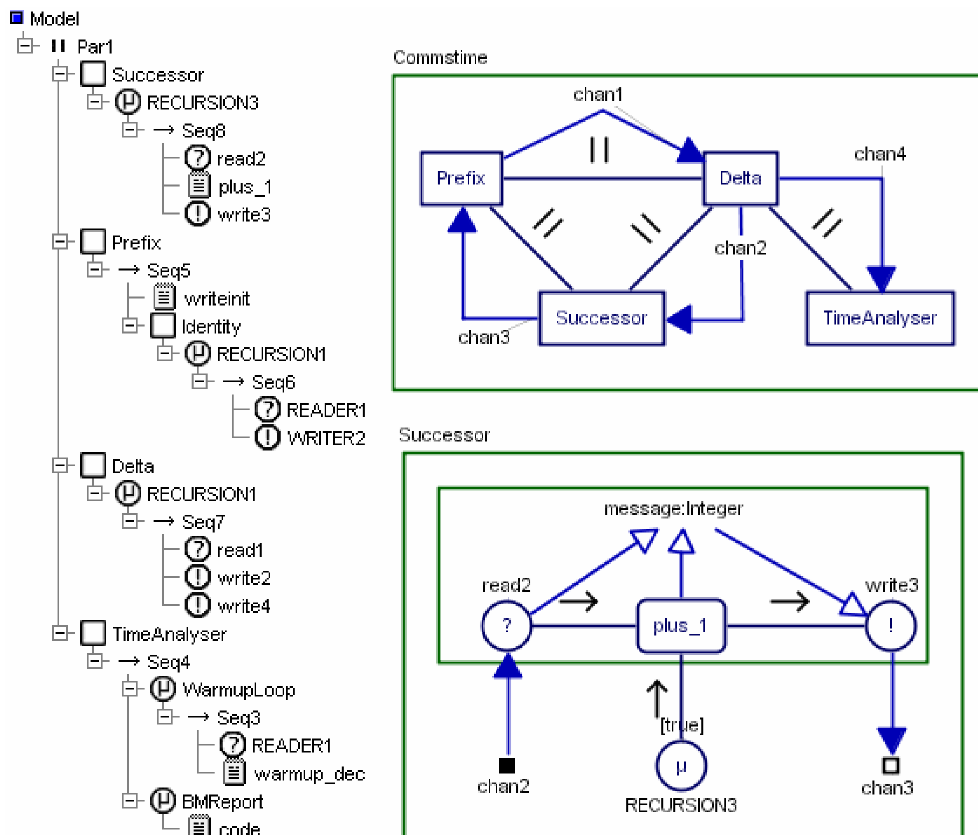

Figure 4: Commstime example in gCSP: composition tree (left) and block diagram (right)

The occam code for this diagram is generated using the new occam code generation output. The generated code is shown in **Listing 2**. All processes are now generated without *inlining*, which results in five sub processes (including the `Identity` subprocess) and a main body with the four *commstime* processes in parallel. Note that the `TimeAnalyser` process contains only the initial warm-up loop. The occam display code for this process will be added further down, because it cannot be drawn completely in gCSP.

```
PROC gCSPModel(CHAN BYTE screen)
  PROC TimeAnalyser(CHAN INT chan4, CHAN BYTE screen)
    INT value:
    INT t0:
    INT t1:
    INITIAL INT looptime IS 100000 :
    INITIAL INT warmup IS 16 :
    SEQ
      WHILE warmup > 0
        SEQ
          chan4 ? value
          warmup := warmup - 1
  :
  PROC Successor(CHAN INT chan3, CHAN INT chan2)
    INT message:
    WHILE TRUE
      SEQ
        chan2 ? message
        message := message + 1
        chan3 ! message
  :
  PROC Delta(CHAN INT chan1, CHAN INT chan4, CHAN INT chan2)
    INT n:
    WHILE TRUE
      SEQ
        chan1 ? n
        chan2 ! n
        chan4 ! n
  :
  PROC Identity(CHAN INT chan1, CHAN INT chan3)
    INT message:
    WHILE TRUE
      SEQ
        chan3 ? message
        chan1 ! message
  :
  PROC Prefix(CHAN INT chan1, CHAN INT chan3)
    INITIAL INT message IS 0 :
    SEQ
      chan1 ! message
      Identity(chan1, chan3)
  :
  CHAN INT chan1:
  CHAN INT chan4:
  CHAN INT chan3:
  CHAN INT chan2:
  PAR
    Successor(chan3, chan2)
    Prefix(chan1, chan3)
    Delta(chan1, chan4, chan2)
    TimeAnalyser(chan4, screen)
:
```

Listing 2: Generated occam 3 code for the *commstime* benchmark using normal generation.

To show the *commstime* statistics on the screen a code block was added to the `TimeAnalyser` process with occam code to display the results every loop. This is done in gCSP using the code dialog window shown in **Figure 5**.

This figure shows the gCSP code dialog that can be used to add (optional) code to a CSP process. Currently CTC++ code, CSPM code and occam code can be added. It is possible to add code for all three languages at the same time, resulting in one gCSP model that can be used for multiple languages.

The missing occam code for displaying the statistics has been copied from the *consume* process in the original KRoC *commstime* example. After the addition of the display code, the generated occam code is comparable with the original *commstime* code delivered with the KRoC compiler.

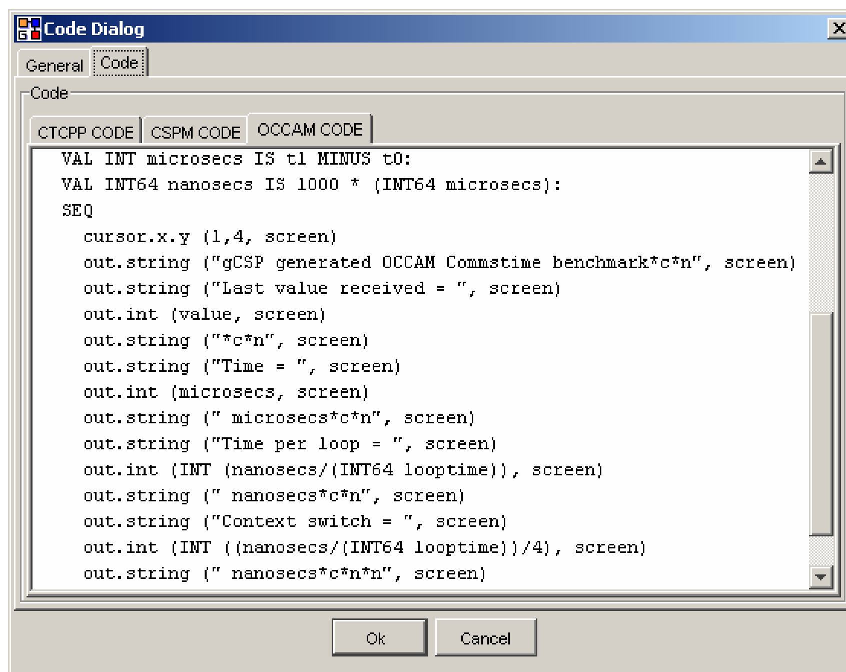

Figure 5: Code dialog for the `TimeAnalyser` showing part of the missing occam code

The occam code block from **Figure 5** uses support functions (`out.string`, `cursor.x.y`) which are not regularly available in occam libraries. For RMoX, `out.string` can be found in the `occ_utils` library. However, `cursor.x.y` is not available. This is solved by adding the missing occam support code to the top level model. The generated occam code is now complete and compilation, using the KRoC compiler, and running under Linux gives a working *commstime* example with a looptime of 990 ns and a context switch of 248 ns (on a 400 MHz Pentium II).

The next step is to get the program running under RMoX. The current version of RMoX (version 0.1.3) has no provision for loading and executing programs. All functionality is included in the RMoX kernel. To run the generated code under RMoX, the RMoX kernel has been extended with an additional console process that contains the generated gCSP program. After code generation, the RMoX operating system has to be (re)compiled with KRoC in order to get the gCSP program running under RMoX. The result is shown in **Figure 6**. The benchmark results under *User Mode* RMoX are doubled (probably due to the emulation). Running under native RMoX gives almost identical results compared to the example running under normal Linux.

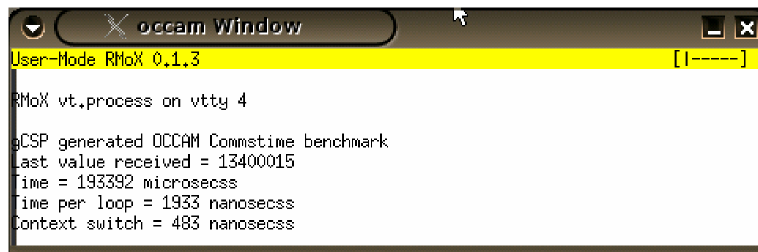

Figure 6: *User-Mode* RMoX running the gCSP Commstime demo

4 Conclusion and Future Work

The construction of a first version of an occam code generator output for gCSP has been successful. gCSP is now able to generate occam code from its models. This code can be compiled with KRoC and runs under RMoX and Linux.

We expect a future use of RMoX on our distributed mechatronic setups, where PC/104 PCs are used as control computer in a hardware-in-the-loop simulation setting. Furthermore, the work on using RMoX gives us the idea for a lean-and-mean embedded OS variant of our CTC++ library and it makes performance comparisons between a minimal operating system and our existing CTC++/RT-linux systems in our control laboratory possible.

Before RMoX can be used for control systems, interfaces from occam code to our I/O hardware devices are needed. occam ports or the new KRoC C-interface can be used for constructing these device drivers. Furthermore, accurate timing support is necessary for control systems to fulfil the requirement of practically jitter-free equidistant time stamps for sampling [10]. RMoX provides a way of blocking an occam process waiting for a hardware interrupt. This can be used to unblock controller processes waiting for a timer interrupt

The tests used here, used a significant portion of occam code in the code blocks. Not every part of an occam program can be drawn completely in gCSP. For example, the use of constants, shared channels and FOR loops is not yet supported in gCSP. This should be added in future versions. Besides this, syntax highlighting in the gCSP code blocks is a useful addition.

References

[1] F.R.M. Barnes, occwserv: An occam Web-Server, in *Communicating Process Architectures 2003*, J.F. Broenink and G.H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 251-268, ISBN: 1 58603 381 6.

[2] F.R.M. Barnes, C. Jacobson, and B. Vinter, RMoX: A raw-metal occam Experiment, in *Communicating Process Architectures 2003*, J.F. Broenink and G.H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 269-288, ISBN: 1 58603 381 6.

[3] F.R.M. Barnes and P.H. Welch, Communicating Mobile Processes, in *Communicating Process Architectures 2005*, I.R. East, J.M.R. Martin, P.H. Welch, D. Duce, and M. Green, Eds. Oxford, UK: IOS Press, 2004, pp. 201-218, ISBN: 1586034588.

[4] A.L. Lawrence, Overtures and hesitant offers: hiding in CSPP, in *Communicating Process Architectures 2003*, J.F. Broenink and G.H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 97-109, ISBN: 1 58603 381 6.

[5] V. Raju, L. Rong, and G.S. Stiles, Automatic Conversion of CSP to CTJ, JCSP, and CCSP, in *Communicating Process Architectures 2003*, J.F. Broenink and G.H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 63-81, ISBN: 1 58603 381 6.

[6] M. Schweigler, F.R.M. Barnes, and P.H. Welch, Flexible, Transparent and Dynamic occam Networking With KRoC.net, in *Communicating Process Architectures 2003*, J.F. Broenink and G.H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 199-224, ISBN: 1 58603 381 6.
[7] D.S. Jovanovic, B. Orlic, G.K. Liet, and J.F. Broenink, gCSP: A Graphical Tool for Designing CSP systems, in *Communicating Process Architectures 2004*, I. East, J. Martin, P.H. Welch, D. Duce, and M. Green, Eds. Oxford, UK: IOS press, 2004, pp. 233-251, ISBN: 1586034588.
[8] G.H. Hilderink, Graphical modelling language for specifying concurrency based on CSP, *IEE Proceedings Software*, vol. 150, pp. 108-120, 2003.
[9] G.H. Hilderink, *Managing Complexity of Control Software through Concurrency,* PhD thesis, University of Twente, Netherlands, 2005, ISBN: 90-365-2204-8.
[10] G.H. Hilderink and J.F. Broenink, Sampling and timing a task for the environmental process, in *Communicating Process Architectures 2003*, J.F. Broenink and G.H. Hilderink, Eds. Enschede, Netherlands: IOS Press, 2003, pp. 111-124, ISBN: 1 58603 381 6.