26th ABCM International Congress of Mechanical Engineering
November 22-26, 2021. Florianópolis, SC, Brazil

COBEM2021-0441
TRANSPILATION FROM NC FILES TO CANONICAL MACHINING FUNCTIONS

Francisco Ricardo Taborda Aguiar
Dalberto Dias da Costa
UFPR - Universidade Federal do Paraná, Av. Cel. Francisco H. Dos Santos, 100 - Jardim das Américas, Curitiba - PR
Programa de Pós-Graduação em Engenharia de Manufatura - UFPR - francisco.ricardo@ufpr.br
Departamento de Engenharia Mecânica - UFPR - dalberto@ufpr.br

Abstract. *The NC file provides the commands for the spindle movements and other events during the machining into CNC machines. But extracting information from the NC file requires a standard format. The NC programming is based on a standard called RS274D which was adopted by ISO and named ISO6893. The CNC manufacturers have extended this NC language by including new commands or changing the syntax of some existent commands to adapt them with their customized functions. The differences in the syntax make it difficult to extract data from the NC file. The present paper proposes an algorithm for interpreting the NC file and transpiling it to Canonical Machining Functions, which is a neutral format to represent the NC data. Also, from the Canonical Machining Functions file is possible to calculate the path, the speeds, the feeds, the tool changes, and the timestamp for these events during the NC file processing. The designed algorithm works like a compiler in computer science. A compiler translates a high-level language into another low-level language. The proposed parser, in turn, translates an NC file into Canonical Machining Functions. The algorithm is based on a two steps flow, like a pipeline. The first step consists of lexical analysis. It splits the NC file into tokens using regular expressions. The second step consists of a semantical analysis. It parses the tokens using grammar rules and recognizes the sentences to generate the Canonical Machining Functions. A prototype has been built to parse NC files created for a specific machine. The regular expressions and the grammar were created based on the CNC programming manual provided by the manufacturer. The algorithm was able to successfully recognizes the G-codes and translates them to Canonical Machining Functions. This approach can solve the lack of interoperability between different NC systems. By adjusting the regular expressions and the grammar rules it is possible to transpile from different input formats and generate the same target format, i.e. Canonical Machining Functions. For future work, a parser can be built for interpreting the Canonical Machining Functions file and extracting information from it.*

Keywords: *RS274-D, ISO6893, G-code, TRANSPILING*

1. INTRODUCTION

In the sixties, the Electronic Industry Association developed a standard called RS274 to program CNC (Computerized Numerical Control) machines. The latest release of the standard dates from 1979 and is called RS274-D. In 1982 this standard was adopted by ISO being called ISO6893. Some countries in Europe have adopted extensions for the standard. For example, the DIN66025 Standard, which is used in Germany. Guo *et al.* (2012) note that the standards are only nominal nowadays as the technology of the CNC systems has advanced a lot since the standards were published. Many functions, not supported by the standards, were added to the CNC systems. Furthermore, the machine vendors have extended the CNC programming standards with additional syntaxes and commands to adapt them to the customized functions. Nowadays, according to Zhang *et al.* (2015), these differences in syntax lead to a so-called *NC dialects* instead of a standard *NC language*. These dialects impose a barrier to the interoperability and integration of the NC programs between different systems. So, different programs must be generated for machining the same part in different machines. The lack of interoperability affects the manufacturing information flow in two manners. Firstly, code changing that always happen at the shop floor cannot be propagated upstream, i.e., to other systems, such as CAPP (Computer Aided Process Planning), CAM (Computer-Aided Manufacturing), and CAD (Computer-Aided Design). Secondly, systems located downstream, such as QIF (Quality Information Framework) and MES (Manufacturing execution systems) will demand specialized interfaces to capture the changes that affect, for example, dimensions or the processing time. The main focus of this work is to deal with the second problem before mentioned, i. e., the lack of interoperability that happens downstream in the machining information flow just after the generation and processing of the numerical control programs.

According to Chen *et al.* (2008), the concept of interoperability differs from integration. In their view, integration means that all components are "tightly coupled" which indicates that the components are interdependent and cannot be

separated. On the other hand, interoperability means components "loosely coupled" in which they can exchange services while continuing locally their logic of operation. Based on that concept of "coupling", Chen *et al.* (2008) conclude that "Thus two integrated systems are inevitably interoperable, but two interoperable systems are not necessarily integrated".

Integration and interoperability are key concepts in modern manufacturing systems. As pointed out by Adamczyk *et al.* (2020) the new smart manufacturing industry must employ an extensive range of software tools to support its manufacturing processes, which imposes a great challenge in terms of interoperability and integration.

However, the digital manufacturing flow related to a product life cycle only happens in a unidirectional way from the design to the shop floor (downstream). The upstream flow, i. e., the feedback from shop floor to design is not supported yet.

A great effort to face the problem with interoperability and integration has been carried out by the ISO Technical Committee 184 - Industrial automation systems and integration - and has been formalized in the ISO 10303 standard "Industrial automation systems and integration - Product data representation and exchange" also known as STEP.

The data transfer between CAD/CAM systems has been contemplated by the ISO Technical Committee 184 in two approaches: as the application reference model (ARM) version of ISO14649 and as the Application Interpreted Model (AIM) a version of ISO14649, which is documented in the Application Protocol 238. Both the approaches are known as STEP-NC (Kumar *et al.*, 2007).

Many papers have been written aiming at the use of the STEP-NC standard to integrate the information flow on the machining process. In Jaider *et al.* (2015), for example, the authors propose the building of a decision system for the selection of cut tools. The selection occurs automatically from the machining features which are extracted from the STEP-NC file through a program written in Python. Álvares *et al.* (2020) developed six different STEP-NC architectures for robotic machining applications.

The lack of monitoring and traceability capability in the STEP-NC was addressed by Campos and Miguez (2011). Danjou *et al.* (2017) proposed a closed-loop manufacturing approach to achieve the information feedback from CNC machines to CAM systems based on STEP-NC. Dharmawardhana *et al.* (2018) review the research and development of STEP-NC controllers in the last decade and their capabilities to implement closed-loop interoperability.

There are two complementary efforts reported in the literature which deal with the apparent lack of monitoring capability observed in STEP-NC. The first one is the MTConnect Standard. MTConnect is a data and information exchange standard that is based on a data dictionary of terms describing information associated with manufacturing operations (MTConnect-Institute, April 2021). The second standard is the OPC-UA (Open Platform Communications - Unified Architecture). The OPC-UA 40501 specifies an Information Model for the representation of a machine tool, providing conditions to allows an exchange of information between a machine tool and software systems like MES, SCADA, ERP, or data analytics systems (OPC-Foudation, April 2021).

As pointed out by Liu *et al.* (2018) and Navas *et al.* (2021) this set of standards beforementioned is a necessary condition to solve the lack of interoperability and integration in the current CNC machining and promote the implementation of smart manufacturing based on Cyber-Physical Machine Tools.

However, machining processes still rely upon old machines controlled by proprietary NC systems which, in the words of Zhang *et al.* (2015), are considered dialects of the EIA 274D or ISO 6983 standards. In this context, one intermediary layer, between the CNC controller and a high-level standard, such as STEP-NC for instance, needs to be developed.

Several years ago, the Intelligent Systems Division (ISD) of the National Institute of Standards and Technology (NIST) worked on the Enhanced Machine Controller (EMC) project (Kramer and Proctor, 1998). The purpose of that project is to build application programming interface standards for open-architecture machine controllers and demonstrate implementations of the Next Generation Controller (NGC) architecture. The NGC architecture has independent parts, one of which is a specification for the RS274/NGC language, an extension of the RS274-D standard. The EMC project is working with industrial partners in an open-architecture machine tool controller known as VGER. The work of Kramer and Proctor (1998) describes the RS274/VGER interpreter. It is a system that reads numerical control code in the RS274/NGC dialect and produces calls to a set of canonical machining functions. The interpreter has two modes: integrated with the EMC controller and stand-alone. The EMC control system interprets either NC codes from files or individual commands entered using the manual data input capability of the control system. In stand-alone mode, the interpreter reads codes either from the keyboard or from an NC file and prints the canonical machine functions to the computer terminal but may be redirected to a file. The authors defined the production rules and the tokens for modeling the parse tree. The interpreter is written in C++ language. But it is focused on a specific NC file format, the RS274/NGC.

The Canonical Machining Functions were defined by the National Institute of Standard and Technology. The authors in Proctor *et al.* (1997) explain that the Canonical Machining Functions were designed with three objectives in mind. First, all the functionality of 3-axis to 6-axis machining centers had to be covered by the commands. Second, portability for using available commercial motion control boards from various vendors to carry out the canonical commands which call for motion. Third, it must be possible to interpret RS274 commands into Canonical Machining Functions. The Canonical Machine Functions are atomic commands. It means that each command produces a single tool motion or a single logical action. The RS274 commands, in turn, include two types. Those for which a single RS274 command corresponds exactly

to a Canonical Machine Function. And those for which a single RS274 command will be translated into several Canonical Machine Functions.

Guo *et al.* (2012) propose a universal NC program processor for CNC systems, called NCPP (Numeric Control Program Processor). Different NC file specifications are stored in dictionaries called NCSD (Numeric Control Specification Dictionary). A separated engine was designed for reading NC files and checking the syntax. The engine works like language compilers. A language compiler translates programs written in a source language into an equivalent program in an object language. In Aaby (2003) the author says that a typical compiler consists of chained steps and the output of each step is passed to the next one. The following sequence illustrates the main phases for a compiler.

- The lexical analysis splits the input into lexical groups or tokens. The tokens are defined by regular expressions. The lexical analyzer is also called scanner and is implemented as a finite state machine.
- The parser groups tokens into syntactical units. The parser is implemented as a push-down automata. The program structure recognized by a parser is defined using context-free grammars. The output of the parser is a parse tree representation of the program.
- The semantic analysis analyzes the parse tree. Information about variables and other objects is stored in a symbol table for checking. Usually, this phase is combined with the parser.
- The code generator transforms the parse tree into object code using rules. The code generator is often combined with the parser.

The methodology proposed by Guo *et al.* (2012) uses transpiling to translate the NC files. Transpiling (or transcompiling, or source-to-source compilation) refers to the process of translating source code written in one language into another language. So transcompiler is a special kind of compiler. The language compilers generate a lower-level code (usually a machine code). The transcompilers, in turn, generate a source code with a similar level of abstraction. Transpilation has been used in many industrial applications. Emscripten (Zakai, 2011) allows the translation of C/C++ code into Javascript code for the execution of applications in modern web browsers. Nunnari and Heloir (2018) propose a method to implement once the motion controllers of virtual humans in a reference implementation language and transpile it to multiple game engines (e.g. C#, Python, and so on). In Maheshwari and Reddy (2015) the authors present an approach to transpile legacy ActionScript3 code into Javascript. Bouraqadi and Mason (2016) use transpilation to convert SmallTalk code into Javascript code. Bysiek *et al.* (2016) use transpilation as an approach to convert Fortran code into modern Python code.

The engine designed in Guo *et al.* (2012) has three modules: lexical analysis, syntax analysis, and canonical machine functions generator. The NC program syntax is represented by a variant of BNF (Backus Naur Form). BNF is a meta-language, that syntactically describes a programming language (McCracken and Reilly, 2003). The authors of Naur *et al.* (1997) used this technique to describe the ALGOL60 language. A BNF grammar can be used to express context-free grammars. Context-free grammar has two sets of symbols. The first is the set of nonterminal symbols. The other one is the set of terminal symbols or tokens. The sentences of the language generated by a grammar will contain only terminal symbols (Aho and Johnson, 1974). A context-free grammar consists of a finite set of rules called productions, expressed by the following form: $\mathbf{LHS}\rightarrow\mathbf{RHS}$, where $\mathbf{LHS}$ (left-hand side) is a single nonterminal symbol and $\mathbf{RHS}$ (right-hand side) is a string of zero or more grammar symbols. In Johnson and Sethi (1990) we see that a production defines a nonterminal in terms of a sequence of terminals and nonterminals. The authors in Aho and Johnson (1974) define a grammar as a rewriting system. If $\alpha\mathbf{A}\gamma$ is a string of grammar symbols and $\mathbf{A}\rightarrow\beta$ is a production, then we may write the following:

$$\alpha\mathbf{A}\gamma\Rightarrow\alpha\beta\gamma \tag{1}$$

We may say that $\alpha\mathbf{A}\gamma$ directly derives β.

The authors in Guo *et al.* (2012) implemented one interpreter, in TCL language, for each command. The mechanism translates the NC file into Canonical Machine Functions if no syntax error was detected during the file interpretation. The main benefit of the designed method is the flexibility for processing different types of NC files, being necessary only to adjust the NCSD module for translating new files. But the authors preferred to create the syntax analyzer by hand, using technologies like TCL, which increases the complexity for the implementation.

Schroeder and Hoffmann (2006) propose an approach for assisting the user by converting NC files from one standard to another. The authors implemented a software tool based on regular expressions. The regular expressions are stored in an external file, in XML format. Each line of the NC file is parsed by the software to find the tokens defined in the regular expressions. A regular expression specifies a set of strings to be matched in a search pattern. The concept arose in the fifties when the mathematician Stephen Cole Kleene developed a formal description of a regular language (Kleene, 1951). It is widely used in Unix text-processing utilities, like Awk (Aho *et al.*, 1979). A regular expression can be expressed as a Finite State Automaton which can be represented using states, and transitions between states. There is one start state, and one or more final or accepting states. Let us consider the regular expression *"[0-9]+"*. It matches only for digits. In Fig. 1, state 0 is the start state, and state 2 is the accepting state.

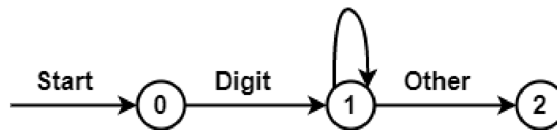

Figure 1: Finite State Automaton.

The Finite State Automaton transitions from one state to another while the characters are read from the input. As the first digit is read the Finite State Automaton moves to state 1. It remains in state 1 while more digits are read. When it reads a character different from a digit it moves to state 2 (the accepting state). A Finite State Automaton may be expressed as a computer program. In the approach proposed by Schroeder and Hoffmann (2006), when a regular expression is matched with a token, it is removed from the processed line to avoid loops. The not-matched tokens are shown to the user. If necessary, the user can adjust the regular expressions to improve the results. The work presented a simple and flexible solution for converting NC files from one type to another. But the algorithm is based only on regular expressions and it can not parse multi-line commands.

2. METHODOLOGY

The present paper proposes an algorithm for interpreting the NC file and transpile it into Canonical Machining Functions. The designed algorithm works as a language compiler in computer science. The source language is the NC file. And the object language is the Canonical Machine Functions file. It is based on a two steps flow.

The first phase, called the Scanner module, reads the input and translates the strings into tokens. A tool called Lex is used for the implementation of the Scanner. Lex was designed for the lexical processing of character streams (Lesk and Schmidt, 1975). and generates a program in C language for a scanner. It uses regular expressions to define patterns that match strings in the input and converts the strings into tokens. The regular expressions are specified in the source specifications given to Lex. The regular expressions are translated by Lex to a computer program in language C that works like a Finite State Automaton. The time taken by a Lex program to process an input stream is proportional to the length of the input. The number of the rules is not important in determining speed.

Although Lex is a good tool for pattern matching, it can not deal with nested data because this kind of data requires a stack data structure to push and pop data as the patterns are matched. However, Lex only has states and transitions between states. But another tool, called Yacc, has a stack and is the right tool for handling nested structures. So, the second module, called Parser, is based on Yacc.

Yacc is is a tool for building syntax analyzers (Johnson and Sethi, 1990). It has been used to implement computer languages since the seventies. It generates parsers from a specification language that describes the desired syntax. A Yacc specification consists of a collection of grammar rules that describes the syntax of a language (Johnson and Sethi, 1990). The grammar for Yacc is specified using a variant of BNF. Yacc turns the specification into a parser. The parser generated by Yacc is based on LR parsers and consists of a finite-state machine and a stack (Johnson, 1980). An LR parser is a shift-reduce parser algorithm that works by scanning the input from left to right. It has the following actions: shift, reduce, accept, and error. The parser reduces an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storage.

Let us consider the NC block *X-15.0 Y-65.0 Z3.0*. The tokens (terminals) are returned by Lex, according to the following regular expressions:

X = "[X][−+]?[0−9]+[.]?[0−9]∗"
Y = "[Y][−+]?[0−9]+[.]?[0−9]∗"
Z = "[Z][−+]?[0−9]+[.]?[0−9]∗"

The productions are specified in Tab. 1.

Table 1: Productions for the example.

#	Production
1	E → X Y Z E
2	E → X Y E
3	E → X Z E
4	E → X E
5	E → Y Z E
6	E → Y E
7	E → Z E
8	E → \n

The terms that appear on the left-hand side (LHS) of a production, such as E (expression) are nonterminals. The terms X, Y, and Z are terminals (tokens returned by Lex) and only appear on the right-hand side of a production. At each step the algorithm expands a term, replacing the LHS of a production with the corresponding RHS. Table 2 demonstrates the parsing of the string.

Table 2: Shift-reduce parsing for the example.

Step	Input scanning	Action
1	. X-15.0 Y-65.0 Z3.0 \n	Shift
2	X-15.0 . Y-65.0 Z3.0 \n	Reduce (production 4)
3	E . Y-65.0 Z3.0 \n	Shift
4	Y-65.0 . Z3.0 \n	Reduce (production 6)
5	E . Z3.0 \n	Shift
6	Z3.0 . \n	Reduce (production 7)
7	E . \n	Shift
8	\n .	Reduce (production 8)
9	E .	Accept

The terms to the left of the dot are on the stack, while the remaining input is to the right of the dot. It starts by shifting tokens onto the stack. When the top of the stack matches the RHS of a production, it replaces the matched tokens on the stack with the LHS of the production. Conceptually, the matched tokens of the RHS are popped off the stack, and the LHS of the production is pushed on the stack. The matched tokens are known as a handle, and we are reducing the handle to the LHS of the production. This process continues until we have shifted all input to the stack, and only the starting nonterminal remains on the stack. In step 1 it shifts the token X to the stack. In step 2 applies production 3 to the stack, changing X to E. Parsing actions are done when a production is reduced. This process continues until all the input has been scanned and the stack contains only the start symbol or an error has been encountered.

The implementation used Flex, a faster version of Lex (Aaby, 2003), and Bison, a faster version of Yacc (Aaby, 2003). Bison expects its lexical analyzer function to be named *yylex()*. So Flex uses that name to interface with Bison. Figure 2 demonstrates the compiling and linking of Bison and Flex to work together.

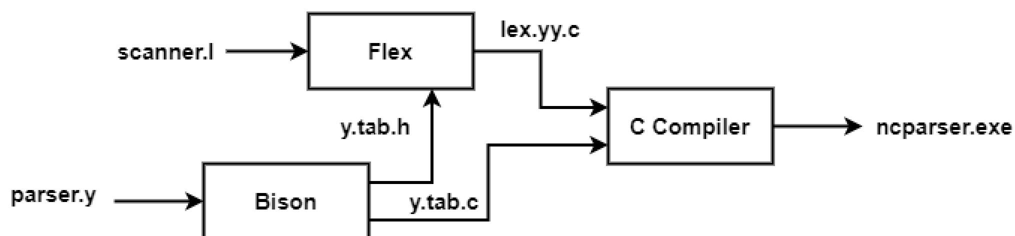

Figure 2: Flex an Bison linking.

The Scanner is implemented by Flex and the Parser is implemented by Bison. Figure 3 gives an idea about the association between the Scanner and the Parser.

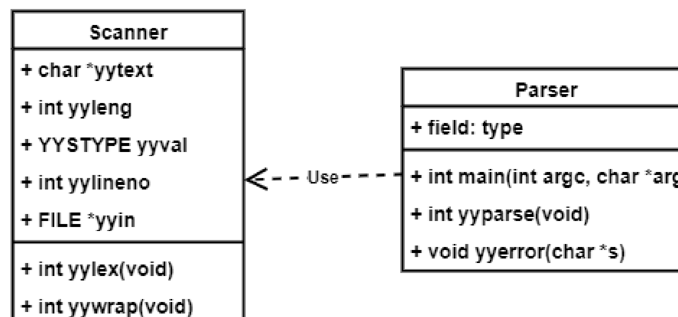

Figure 3: Flex an Bison association.

The Scanner returns a stream of tokens that are handled by the Parser. Each time the Parser needs a token, it calls *yylex()*, which reads a portion of the input and returns the token. When it needs another token, it calls *yylex()* again. The Scanner acts as a coroutine (Lewis, 2003). Each time it returns, it remembers where it was, and on the next call it picks

up where it left off (Levine, 2009). Within the Scanner, when a token is ready, it returns it as the value from *yylex()*. The next time the Parser calls *yylex()*, it resumes scanning with the next input characters. If a pattern does not produce a token for the Parser, the Scanner will keep going within the same call to *yylex()*, scanning the next input characters.

The algorithm is conceptually represented in fig. 4.

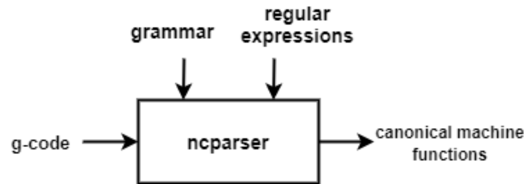

Figure 4: Functional concept.

The input is an NC file that is transpiled to Canonical Machine Functions. The regular expressions and the grammar rules are defined to parse a specific G-code standard. They rule the transpilation process by controlling the scanning and the parsing of the input file. By adjusting the regular expressions and the grammar rules it is possible to parse different input formats, i.e different G-code standards, and generate the Canonical Machining Functions.

The output of the parser can be passed directly as input to the next parser to get more functionality. It resembles the UNIX pipelines (Ritchie, 1984). For example, it is possible to build a post-processor to parse the Canonical Machining Functions and transpile them into different G-code standards. Figure 5 demonstrates the pipeline for this purpose. The input for the first parser is the NC file and the output is the Canonical Machining Functions. The second stage (post-processors) receives the Canonical Machining Functions and transpiles them into different G-code standards. This approach can deal with the interoperability issue.

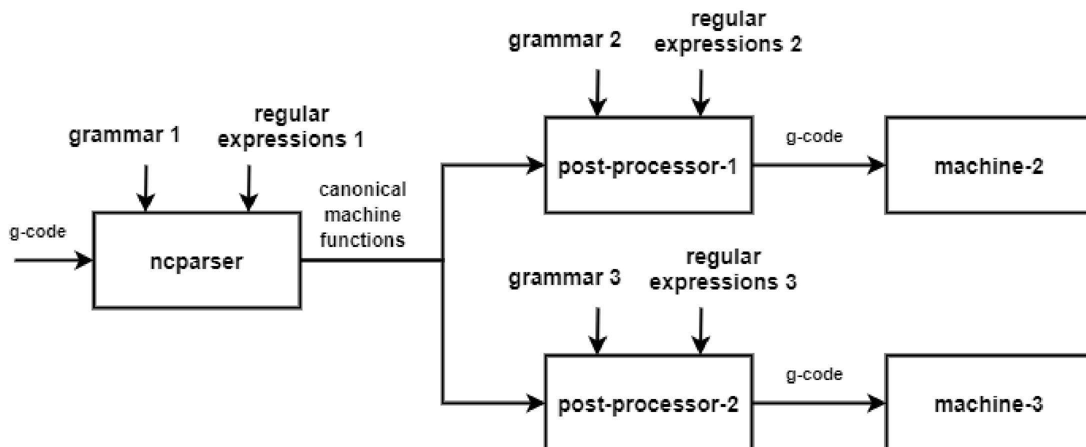

Figure 5: Post-processor to deal with interoperability issues.

3. CASE STUDY, RESULTS AND DISCUSSIONS

The proposed methodology was prototyped in C, using Flex for the scanner and Bison for the parser. The before mentioned prototype was tested in an old CNC controller (Mach-9 from Romi - Brazil) available in a three-axis machining center (Discovery from Romi - Brazil). The Mach-9 controller is almost adherent to EIA-274D. A simple test specimen (see Fig. 6) was designed to be machined in a single set-up by face milling and drilling operations with canned cycles.

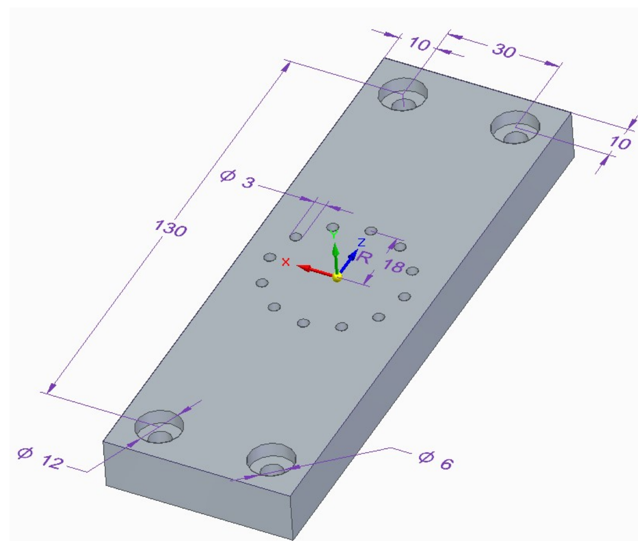

Figure 6: Machining test.

The NC program was generated in a CAM system (Edgecam 2019 R2 from Hexagon - Sweden), which has a specific postprocessor for Mach-9.

A partial view of the native NC program (Mach-9) is shown in Tab. 3, where the first column contains the original G-code (Mach-9) and the second contains the transpiled Canonical Functions. It can be observed that canned cycles G81 and G25 were converted to a sequence of “straight_feed” and “straight_traverse” commands.

Table 3: Transpiling of the second NC file.

NC File Block	Canonical Machine Function
T3M6	SELECT_TOOL(3) CHANGE_TOOL(3)
O3S1857M3	SET_SPINDLE_SPEED(1857.0) START_SPINDLE_CLOCKWISE()
G0X-15.Y-65.Z3.	STRAIGHT_TRAVERSE(-15.0, -65.0, 3.0)
X-15.Y-65.Z5.	STRAIGHT_TRAVERSE(-15.0, -65.0, 5.0)
G81Z-8.R3.F186 G25X30.Y130.I2J2 G80	STRAIGHT_TRAVERSE(-15.0, -65.0, 3.0) SET_FEED_RATE(186.0) STRAIGHT_FEED(-15.0, -65.0, -8.0) STRAIGHT_TRAVERSE(-15.0, -65.0, 3.0) STRAIGHT_TRAVERSE(15.0, -65.0, 3.0) SET_FEED_RATE(186.0) STRAIGHT_FEED(15.0, -65.0, -8.0) STRAIGHT_TRAVERSE(15.0, -65.0, 3.0) STRAIGHT_TRAVERSE(15.0, 65.0, 3.0) SET_FEED_RATE(186.0) STRAIGHT_FEED(15.0, 65.0, -8.0) STRAIGHT_TRAVERSE(15.0, 65.0, 3.0) STRAIGHT_TRAVERSE(-15.0, 65.0, 3.0) SET_FEED_RATE(186.0) STRAIGHT_FEED(-15.0, 65.0, -8.0) STRAIGHT_TRAVERSE(-15.0, 65.0, 3.0)
M5	STOP_SPINDLE_TURNING()

That decomposition of the canned cycles in more detailed commands is a very interesting characteristic of the Canonical Functions and the transpiling procedure devised here. It is interesting because a more granulate piece of information is derived which allows the further follow-up of the NC program during its execution, independent of its original NC dialect. A synchronized follow-up of the NC program can provide valuable information to compare the programmed speed and feed values with those early planned. The tool changes and their timestamps are also important to estimate and verify the machining lead time. In addition, the Canonical Machining Functions behave like an intermediary layer of abstraction,

which can be used for creating an interface with upstream/downstream systems, such as MES, ERP, and CAPP. Nevertheless, it should be pointed out that, as shown in Fig. 5, for each available NC dialect is necessary to develop a specific parser.

4. CONCLUSIONS

The present paper proposes an algorithm for interpreting the NC file and transpile it into Canonical Machining Functions. The designed algorithm works as a compiler in computer science. It is based on a two steps flow. The first one splits the NC file into tokens using regular expressions. The second step parses the tokens using context-free grammar and recognizes the sentences to generate the Canonical Machining Functions. From the Canonical Machining Functions file is possible to follow up the speeds, the feeds, the tool changes, and the timestamp for these events during the NC file processing. The results already achieved allows us to affirm that the proposed methodology can be extended further by chaining more parsers with different regular expressions and grammars. It is expected that the proposed methodology will contribute to reduce the lack of interoperability observed in the old CNC machines and act as an intermediary layer of abstraction, which can be used for creating an interface with upstream/downstream systems, such as MES, ERP, and CAPP.

5. REFERENCES

Aaby, A.A., 2003. "Compiler construction using flex and bison". *Walla Walla College*.

Adamczyk, B.S., Szejka, A.L. and Júnior, O.C., 2020. "Knowledge-based expert system to support the semantic interoperability in smart manufacturing". *Computers in Industry*, Vol. 115, p. 103161.

Aho, A.V. and Johnson, S.C., 1974. "Lr parsing". *ACM Computing Surveys (CSUR)*, Vol. 6, No. 2, pp. 99–124.

Aho, A.V., Kernighan, B.W. and Weinberger, P.J., 1979. "Awk - a pattern scanning and processing language". *Software: Practice and Experience*, Vol. 9, No. 4, pp. 267–279.

Álvares, A.J., Rodriguez, E., Jaimes, C.I.R., Toquica, J.S. and Ferreira, J.C., 2020. "Step-nc architectures for industrial robotic machining: Review, implementation and validation". *IEEE Access*, Vol. 8, pp. 152592–152610.

Bouraqadi, N. and Mason, D., 2016. "Mocks, proxies, and transpilation as development strategies for web development". In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*. pp. 1–6.

Bysiek, M., Drozd, A. and Matsuoka, S., 2016. "Migrating legacy fortran to python while retaining fortran-level performance through transpilation and type hints". In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE, pp. 9–18.

Campos, J.G. and Miguez, L.R., 2011. "Standard process monitoring and traceability programming in collaborative cad/cam/cnc manufacturing scenarios". *Computers in Industry*, Vol. 62, No. 3, pp. 311–322.

Chen, D., Doumeingts, G. and Vernadat, F., 2008. "Architectures for enterprise integration and interoperability: Past, present and future". *Computers in industry*, Vol. 59, No. 7, pp. 647–659.

Danjou, C., Le Duigou, J. and Eynard, B., 2017. "Manufacturing knowledge management based on step-nc standard: a closed-loop manufacturing approach". *International Journal of Computer Integrated Manufacturing*, Vol. 30, No. 9, pp. 995–1009.

Dharmawardhana, M., Oancea, G. and Ratnaweera, A., 2018. "A review of step-nc compliant cnc systems and possibilities of closed loop manufacturing". In *IOP Conference Series: Materials Science and Engineering*. IOP Publishing, Vol. 399, p. 012014.

Guo, X., Liu, Y., Du, D., Yamazaki, K. and Fujishima, M., 2012. "A universal nc program processor design and prototype implementation for cnc systems". *The International Journal of Advanced Manufacturing Technology*, Vol. 60, No. 5, pp. 561–575. ISSN 1433-3015. doi:10.1007/s00170-011-3618-6. URL `https://doi.org/10.1007/s00170-011-3618-6`.

Jaider, O., El Mesbahi, A., Rechia, A. and Zarkti, H., 2015. "An automatic Feature-based tool selection approach for turning process based on data from Sandvik Coromant". In *Xème Conférence Internationale : Conception et Production Intégrées*. Tanger, Morocco. URL `https://hal.archives-ouvertes.fr/hal-01260833`.

Johnson, S.C., 1980. "Language development tools on the unix system." *IEEE Computer*, Vol. 13, No. 8, pp. 16–21.

Johnson, S.C. and Sethi, R., 1990. "Yacc: a parser generator". *UNIX Vol. II: research system*, pp. 347–374.

Kleene, S., 1951. "Representation of events in nerve nets and finite automata (no. rand-rm-704)". *Rand Project Air Force Santa Monica Ca*.

Kramer, T. and Proctor, F., 1998. "The nist rs274/vger interpreter".

Kumar, S., Nassehi, A., Newman, S.T., Allen, R.D. and Tiwari, M.K., 2007. "Process control in cnc manufacturing for discrete components: A step-nc compliant framework". *Robotics and Computer-Integrated Manufacturing*, Vol. 23, No. 6, pp. 667–676.

Lesk, M.E. and Schmidt, E., 1975. "Lex: A lexical analyzer generator".

Levine, J., 2009. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.".

Lewis, B.T., 2003. "Coroutine". In *Encyclopedia of Computer Science*, pp. 465–466.

Liu, C., Xu, X., Peng, Q. and Zhou, Z., 2018. "Mtconnect-based cyber-physical machine tool: a case study". *Procedia Cirp*, Vol. 72, pp. 492–497.

Maheshwari, Y. and Reddy, Y.R., 2015. "Transformation of flash files to html5 and javascript". In *Proceedings of the ASWEC 2015 24th Australasian Software Engineering Conference*. pp. 23–27.

McCracken, D.D. and Reilly, E.D., 2003. *Backus-Naur Form (BNF)*, John Wiley and Sons Ltd., GBR, pp. 129–131. ISBN 0470864125.

MTConnect-Institute, April 2021. *MTConnect Standard*. MTConnect Institute. URL `https://www.mtconnect.org`.

Naur, P., Backus, J.W., Bauer, F.L., Green, J., Kafz, C., McCarthy, J., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B. *et al.*, 1997. "Revised report on the algorithmic language algol 60". In *ALGOL-like Languages*, Springer, pp. 19–49.

Navas, C.F.E., Yepes, A.E., Abolghasem, S. and Barbieri, G., 2021. "Mtconnect-based decision support system for local machine tool monitoring". *Procedia Computer Science*, Vol. 180, pp. 69–78.

Nunnari, F. and Heloir, A., 2018. "Write-once, transpile-everywhere: re-using motion controllers of virtual humans across multiple game engines". In *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*. Springer, pp. 435–446.

OPC-Foudation, April 2021. *OPC Unified Architecture*. OPC Foundation. URL `https://reference.opcfoundation.org/v104/MachineTool/v100/docs`.

Proctor, F.M., Proctor, F.M. and Michaloski, J.L., 1997. "Canonical machining commands".

Ritchie, D.M., 1984. "The unix system: The evolution of the unix time-sharing system". *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pp. 1577–1593.

Schroeder, T. and Hoffmann, M., 2006. "Flexible automatic converting of nc programs. a cross-compiler for structured text". *International Journal of Production Research*, Vol. 44, No. 13, pp. 2671–2679. doi: 10.1080/00207540500455841. URL `https://doi.org/10.1080/00207540500455841`.

Zakai, A., 2011. "Emscripten: an llvm-to-javascript compiler". In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. pp. 301–312.

Zhang, X., Nassehi, A. and Newman, S.T., 2015. "A meta-model of computer numerical controlled part programming languages". *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, Vol. 229, No. 7, pp. 1243–1257.

6. RESPONSIBILITY NOTICE

The authors are solely responsible for the printed material included in this paper.