

EMS – A Workflow Programming Language and Environment

George Pashev¹, George Totkov²

¹University of Plovdiv “Paisii Hilendarski”, Tsar Assen 24 Str., Plovdiv, Bulgaria

Abstract – The current paper includes formal definition of the grammar of the proprietary scripting procedural language used by the proprietary EMS Workflow management Environment of Dextro Research Ltd. (DR) as an inseparable part of project metadata, which developers create using the DR Script Editor and DR Step Shop IDE tools. Unique features and operators of the language, such as Flow Identifier are defined and example usages are depicted and explained. The paper describes an innovative method of process steps addressing using flow identifiers and position context and current step number context. The trigger subsystem is described, the types of triggers of various events are listed, and examples are given. Output documents generation mode of the interpreter is functionally described and special built-in operators in the language grammar and global variables, aim to make the process as easy as possible.

Keywords – workflow, flow identifier, programming language, flow operators.

1. Introduction

In order to build a better Workflow programming environment and language, certain research questions have to be taken into account: *Q1. How can the*

amount of code definitions and usages be minimized; Q2. Is it possible to combine various mathematical paradigms such as polar coordinate system in favor of Q1; Q3. Is it possible to design and implement an intuitive and easy to use flow addressing mechanism and language operator; Q4. How easy is for the workflow system to support human tasks; Q5. How human tasks can be more universally defined; Q6. How a flow-programming environment can be implemented in such a way, that semantic errors can be avoided on the development stage. Q7. Can flow programming project metadata be structured in such a way, that end user GUI can be automatically generated.

Many of the reviewed well-known programming languages especially designed for workflow applications do not fully meet the requirements assumed by these research questions. Many languages rely on rather static workflow patterns [11,10,7]. Many of them [11,10] rely on large XML definitions which make the code rather large and does not allow the usage of some benefits of procedural languages. On the other hand, it has been noted that standard [7] fails to support human tasks, that is, tasks that are allocated to human actors and that require these actors to complete actions, possibly involving a physical performance. Some extensions of [7] and [10] natively support human tasks, but they just support several types of human tasks, which do not adequately meet the requirements of real life human tasks and events. Therefore, it is vitally important for a workflow system to allow developers to define the task types themselves. Many of the languages, such as [1] are graphical data flow languages, which provide GUI interface for algorithm definition, which is somehow user friendly for non-experienced developers, but makes the continuous code support quite difficult; since complex algorithms would need graphs with large amount of nodes and edges. The graphical only functionality leads to “drag and drop” syndrome. The “drag and drop” syndrome leads to a development process in which the developer is too busy to perform “drag and drop” operations, rather than to be able to focus on the pure programming logic. If a developer

DOI: 10.18421/TEM73-21

<https://dx.doi.org/10.18421/TEM73-21>


Corresponding author: George Pashev,
University of Plovdiv “Paisii Hilendarski”, Tsar Assen 24
Str., Plovdiv, Bulgaria

Email: georgepashev@uni-plovdiv.bg

Received: 08 May 2018.

Accepted: 15 August 2018.

Published: 27 August 2018.

 © 2018 George Pashev, George Totkov; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDeriv 3.0 License.

The article is published with Open Access at www.temjournal.com

spends more time on building a graphical scheme issues rather than solving specific flow application issues, then this flow programming language is not worth using. *All of the reviewed systems lack a satisfactory good mechanism for semantic errors avoidance.*

2. Our Approach and Language Overview

In the next paragraphs, we present our approach and try to provide answers to the above presented research questions.

2.1. Q1 & Q4 & Q5 & Q6 & Q7 Project metadata structure and Process definition

The set of all finite tuples of S would be denoted as $Seq(S)$. The zero set e including the zero tuple e .

The set of all finite subsets of S (including the empty set $-\emptyset$) will be denoted as $Set(s)$, a set build of elements g_1, g_2, \dots, g_m with $set(g_1, g_2, \dots, g_m)$ or $\{g_1, g_2, \dots, g_m\}$.

A project Metadata M is formally defined as:

$$M = \langle C, S, P \rangle$$

Where C is Class data, S is Step data, P is process data.

$$C_i = \langle$$

Name, Text, ABBR, Number, BaseClass, FieldType and $C_i \in C$ where $i \in [1, |C|]$. *Name* is a Class Name, which is visible to the developer, *Text* is the text of a current class, which is visible to the end user, *ABBR* is a class abbreviation, which is a unique alias of the class, $BaseClass \in C \cup \emptyset$ and $BaseClass \neq C_i$ and *BaseClass* is a base class of the current class, with which they form class hierarchy. *Number* is a unique class number and $Number \in [1, 998]$. A special class Number is the Number 999, which is a base class number for all of the registered classes in the system. A special class Number is the number 0, which is a base class for none of the classes. *FieldType* is the functional implementation of the user interface of the class. A few native FieldTypes are currently supported: number, date, text, sequence, nomenclature, nomenclature field participating in hierarchy, HTML, XML, image, etc.

Step Data S is a set containing step definitions $S_j = \langle StepName, StepNumber, StepFields, StepWidth, StepHeight, \dots \rangle$, $j \in [1, |S|]$, which are the definitions of various events which may occur in the context of a process P_k and file. *StepName* is visible to the end user name of the step. *StepNumber* is a unique integer identifier of the step. A special step number 999 is a base step number for all of the registered steps. *StepWidth* and *StepHeight* are the size information for the automatic interface generation of the step. *StepFields* is a user interface control information of the classes, participating in the step. $StepField \in StepFields$ and $StepField = \langle C_i, X, Y, W, H, isVisible, Rights \dots \rangle$, where $X, Y, W, H, isVisible$ are the display parameters of user interface controls. *Rights* is a set of user defined rights for insert, update, delete, view privileges for given user groups.

A way to maximize the avoidance of semantic errors during the code implementation is to make the developer define semantic data classes that include not only data type information, and information about the real life usage of the data class. For example, a typical data class in EMS would be "Date of court trial verdict". After the definition of the required semantic classes, logical steps, in which those classes participate, have to be defined. That allows the employment of many IDE features, like data class hints, data class semantic autocomplete features, during the writing of flow operators into the programming code, etc.

For example, if the developer begins to write a flow identifier, all of its possible forms are proposed as a list. If a class is addressed in a step, which does not include it, the built in semantic error discovery mechanism notifies the developer. Intelligent filtration of step numbers, class numbers and possible flow directions is performed based on the information of the current step context. The developer can choose different step context and run the semantic error discovery mechanism again.

The automatic generation of GUI is possible due to the information stored in the steps and classes, including field types, field coordinates, etc.

A step, process and sub-process in the context of EMS can be viewed as human tasks performed by an individual outside the automation of the workflow application. For example a deployment of a court trial expert report is a task, which is performed outside of the system and results with a document: the report itself, which can be uploaded by the expert, along with some interesting data about the expert report, that can be required. As a result, a new step containing the document and some data classes regarding the document will appear in the case file.

The set of all processes p will be denoted as P.

The EMS Workflow *p* is the graph $p = (N, E)$, which consists of the following:

- A) A set of nodes $N = \{N_i; i \in \overline{1, n}\}$, $N_i = (R_i, D_i, VR_i, User_i, F_i, \Gamma_i, p_i, SR_i, SD_i)$, where:
 - $R_i = \{R_{ij}; j \in \overline{1, s}\}; s \in \mathbb{N}$ is a set of nodes (also called signals), and $R_i \in set(O)$;
 - $D_i = \{D_{il}; l \in \overline{1, k}\}; k \in \mathbb{N}$ – a set of produced objects (also called slots), and $D_i \in set(O)$;
 - $VR_i = \{VR_{il}; l \in \overline{1, k}\}; k \in \mathbb{N}$ – a set of user data entered incoming signals (also called user signals), and $VR_i \in set(O); VR_i \subseteq D_i$
 - $User_i = \{User_{il}; l \in \overline{1, k}\}; k \in \mathbb{N}$ a set of users with the right of read/write/update/delete/execute the current step
 - $F_i = \{F_{ip}; p \in \overline{1, \varepsilon}\}; \varepsilon \in \mathbb{N}$ – a set of output data generative functions $F_{ip}: R_i \rightarrow D_i$;
 - $\Gamma_i = \{\Gamma_{i\phi}; \phi \in \overline{1, \psi}\}; \psi \in \mathbb{N}$ – a set of allowing functions, such as $\Gamma_{i\phi}: R_i \rightarrow \{true, false\}$;
 - $p_i = \{p_{ip}; p \in \overline{1, \varepsilon}\}; \varepsilon \in \mathbb{N}; p_i \subset P$; - a set of sub processes, that can be automatically started or started by user upon the execution of the current step
 - $SR_i = \{SR_{ij}; j \in \overline{1, s}\}; s \in \mathbb{N}$ a set of objects, which are generated automatically, during the execution of triggers and $SR_i \in set(O); SR_i \subset R_i$;
 - $SD_i = \{SD_{ij}; j \in \overline{1, s}\}; s \in \mathbb{N}$ is a set of output data objects which are automatically generated and $SD_i \in set(O); SD_i \subset D_i$;
- B) A set of edges $E \subseteq N \times N$.

3. Q1. EMS Language overview

EMS Language is a context-free procedural command language, which is optimized for command interpreter, in order to achieve a greater interpretation/execution speed.[2, 4]. The current implementation of the command interpreter has all of its commands stored in its memory, so the commands are not invoked as separate processes, like typical Operating Systems command processors do.

The formal definition of the grammar of the procedural language, which covers single line parsing, is discussed in this section.

```

%token<name> TOKEN_ID

%token<val> TOKEN_NUMBER

%token<op> TOKEN_OPERATOR

%token<flow_id> TOKEN_FLOW_ID

%start program

/*some code omitted*/

program: statement '\n' { (*(struct
AstElement**)astDest) = $1; };

block:      TOKEN_BEGIN      statements
TOKEN_END{ $$ = $2; };

statements: { $$=0; }

        | statements statement '\n'
{ $$=makeStatement($1, $2); }

        | statements      block '\n'
{ $$=makeStatement($1, $2); };

statement:

        assignment { $$=$1; }

        | whileStmt { $$=$1; }

        | block { $$=$1; }

        | call { $$=$1; }

assignment: TOKEN_ID '=' expression
{ $$=makeAssignment($1, $3); }

expression:

        expression      TOKEN_OPERATOR
expression      { $$=makeExp($1, $3,
$2); }

        |
{ $$=makeExpByName($1); }

        |
{ $$=makeExpByNum($1); }

        |
{ $$=makeExpByFlowId($1); }

call: TOKEN_ID '(' expression ')'
{ $$=makeCall($1, $3); };
    
```

Fig. 1. Part of the BISON definition of the Language Grammar

As much as the development of command line parser and interpreter is relatively easy [5,4,9], because a line loop procedure could be created, which parses and executes the code line by line, the standard Context Free Language Parser and Lexer generation tools: Bison and Flex are used only for single line parsing. A single line of code consists of a command, command with extra parameters passed, or assignment statement, while statement, block statement, calls function or procedure statement. Assignment statement is similar to assignment rules in other languages, so its discussion is not of importance.

4. Q1 & Q2 & Q3 & Q6. The Flow Identifier Operator

A new and unique feature of the current Programming language is its Flow Identifier Operator, which is used as a variable reference, expression, while loop condition (three in one). It is lexically defined as a regular expression as follows:

```
((((( (@ [ 0-9 ] { 6 } ) ) [ B P N F f b S s L l H h ] { 1 } ) { 1 , } ( \ [ [ . @ \ + \ * \ - \ / 0 - 9 A - Z a - z @ _ \ { \ } \ [ \ ] \ ( \ ) ] * , [ . @ \ + \ * \ - \ / 0 - 9 A - Z a - z @ _ \ { \ } \ [ \ ] \ ( \ ) ] * \ } ) { 0 , 1 } ( \ [ [ . @ \ + \ * \ - \ / 0 - 9 A - Z a - z @ _ \ { \ } \ [ \ ] \ ( \ ) ] * , [ . @ \ + \ * \ - \ / 0 - 9 A - Z a - z @ _ \ { \ } \ [ \ ] \ ( \ ) ] * \ } ) { 0 , 1 } ) | ( @ [ 0-9 ] { 3 } ( \ { , [ . @ \ + \ * \ - \ / 0 - 9 A - Z a - z @ _ \ { \ } \ [ \ ] \ ( \ ) ] * \ } ) { 0 , 1 } ( \ [ [ . @ \ + \ * \ - \ / 0 - 9 A - Z a - z @ _ \ { \ } \ [ \ ] \ ( \ ) ] * , [ . @ \ + \ * \ - \ / 0 - 9 A - Z a - z @ _ \ { \ } \ [ \ ] \ ( \ ) ] * \ } ) { 0 , 1 } ) ) ) )
```

Fig. 2. Lexical Definition of Flow Identifier Operator

The Flow identifier operator is used alongside with the current step context and current process context in the current loaded file in the EMS programming environment. [3]

Generally, fields are indicated depending on what class they have and where the data is in the file and and in the field. The current cursor position is decisive for how we want to access the field. SYNTAX OF SELECTORS [6].

A format selector:

- @ CCCSSSD { } []
- (1)
- @ - Prefix of WfIID
- CCC - including specifying class data. If the number is 999 indicates each class;
- SSS - including step code. If the number is 999 specifies each step;
- D - one of directions: F B P N S L H h b f s l

Intuitively, the flow direction can be only two - forward and backward. The purpose of this filter is dependent on the circumstances of substantial importance for the application system. Classes of data are designed precisely for their intended purpose in the application system. This is why standard filtering includes the presence or absence of the class data (See CCC by selector). Different types of steps: i.e., codes of different steps are also essential to the data. There are functions for even more restrictive filtering steps as follow not only the existence of a class of data, but also a value in this class or data belonging to the process.

SETTING RADIUS RANGE AND INDEXED ARRAY OF SELECTORS

@ CCCSSSD {Radius, Assortment} [Row, Column] (2) where:

- { } - Parameters for the radius and multiplicity;
- [] - Parameters for indexed arrays.

Radius - sets the number of consecutive times to move on in the direction (combination of N identical transitions). Assortment - If multiple fields in one step have the same classes, the first fix a value Assortment = 1 or missing parameter. The second with a value Assortment = 2, etc.

In order to understand the functionality of the flow identifier in-depth, a document generation script code fragment is given as example.

```

...
if DocNumber==0
    SystemSet("{current}",
"matrixtype") ;initial document
generation setting - execute for
currently loaded file only

    stop
endif ; ...

for @999999f ; loop for each field
in each step from the beginning to
ending

    if curstep()==1 ;if metadata
type of current step is 1

        print

    ...

    <td>@(str(@001,"DD.MM.YY"))</t
d>

<td><file>@176{,2}[]</file></td> ;f
low identifier short form

    ...

    endprint

    endif

next
print
    
```

Fig. 3. Example Document generation script

The given example generates a document, which describes the current file in an HTML Table form and loops through the file with the for – next loop block. The example FIID does no filtering for the data class and step type, so all data fields are iterated. Then, if the current step being iterated has a tag “1” in the metadata definition, the current date field and a file field are outputted. Some code is omitted and replaced with “...”.

Examples of current work step form of the flow identifier operator.

- @ 125 - the first 125-class field in the current step;

- @ 125 {1} - the first 125-class field in the current step;
- @ 125 {2} - the second field with Class 125 in the current step;
- @ 125 [3] - the third line of the first class field represented as a string 125
- @ 125 {1} [3] - the third line of the first class field represented as a string 125
- @ 125 {1} [3] - the third line of the first class field represented as a string 125
- @ 125 {2} [, 1] - the first column of the second 125 Class field represented as a string.

Flow identifiers resemble polar coordinate systems i.e., have direction and radius. Syntax is adopted for recording standard directions. Syntax reflects the arrangement of the entries one after the other steps in a file that is equivalent to the actual occurrence of events in time.

- F - The next steps of the current cursor position to the end. Satellites are not taken into account, although the position of the cursor can be followed;
- f - The first step in the file. Use lowercase letter most analogous to F. This is first treated as a next step from the beginning of the file;
- B - One of the previous steps, where the term means before a smaller value of the cursor. Satellites are not taken into account, although the position of the cursor may precede them;
- b - The last step in the file. Refers to the last step and not the last satellite. Use lowercase letter most analogous to B. This is treated first step starting from the end of the file, starting from the beginning to the end;
- N - Just the next step, i.e. immediately after the current step. Satellites not taken into account. This looks like F, but necessarily required satellite is not exactly the next step;
- P - Just the previous step, i.e. immediately before the current step.
- Satellites are not taken into account. It looks like a B, but necessarily requires just the previous non satellite step;
- S - Current satellites next satellite, i.e. if the current step is a satellite. After the current step is a satellite, it can only be the next successor of the same parent. All the inheritors of one parent we call orbit. Therefore, the direction of the S lists all inheritors, i.e. lists those successors who remain after (follow the orbit). If the current step is a satellite, that is does not belong to the current orbit, the direction is empty. There is an analogy with the direction F, but applies only within one orbit;
- s - this direction applies only when the current step is satellite, i.e. when the current step is within one orbit. This direction applies when the current

step is the first in orbit and it is needed to move to the beginning of the orbit. „s” in the direction indicated by the current first satellite orbit. The next application direction is transformed into "S";

- L - leads to the first heir to the orbit of the heirs. In this direction, we move to a higher orbit, i.e. lower in the hierarchy;

- l - in the first step of the application leads to the first successor of the orbit, but in the next application is transformed in the direction S;

- H - is intended to provide guidance to a lower orbit, i.e. higher in the hierarchy, but not to the parent and to the next parent step. It should only be used in satellites. If you are currently positioned on the satellite, that direction leads to a step at the end of the orbit. End of the orbit may be a step or satellite higher in the hierarchy. In particular, direction F, N has a similar effect but the current orbit ends in step instead of satellite, i.e. H direction is used when it is not certain that the current orbit is lowest and expect a lower orbit that is higher in the hierarchy;

- h - similar to H, but refers to a step parent;

If the current step is satellite directions, B and P have the following effect. They point to the first non-satellite step "backwards".

Example: Assume that steps used in Figure 4. contain fields with step classes as follows:

- step st11 - Classes 1, 2, 3, 4
- step st12 - Classes 3, 4, 5, 6, 7
- step st17 - grades 6, 7, 8, 9, 10, 11
- step st22 - Classes 122, 113, 114, 1, 2, 3
- step st43 - Classes 11, 22, 33, 44, 55, 66, 77, 88, 99
- step st81 - classes 81, 82, 83
- Step st133 - Classes 122, 113, 114, 1, 2, 3, 11, 12, 13, 14, 15, 16

Full form of selectors' transitions 1 => 10 can be constructed more easily. Use the fact that Class 122 is presented only in step 10.

22: @122999F => 10

This form is convenient because it could be used for other steps in Class 122, if the existence of a common class of 122 speaks of a similar role within the system.

Access of all other steps to step 17 may be constructed equivalence because step 17 is the only one of its kind. Similar effect can be achieved for a class of data 10, present only in step 17.

- 22: @ 999017f => 17
- 81: @ 999017f => 17
- 133: @ 999017f => 17
- 43: @ 999017f => 17
- 61: @ 999017f => 17
- 12: @ 999017f => 17

- 11: @ 999017f => 17 - this applies to each of the steps of type 11 are located respectively at positions 5, 6 and 7.

- 22: @ 010999f => 17
- 81: @ 010999f => 17
- 133: @ 010999f => 17
- 43: @ 010999f => 17

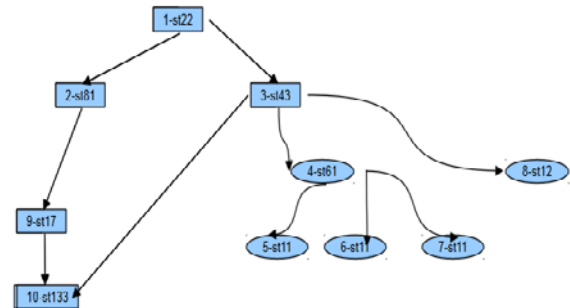


Fig. 4. Example of step and class transitions graph with flow identifier operator

The reader can observe how relatively complex queries can be written with very few lines of code. There is no need of large SQL like queries or XML or JSON query definitions. All the developer has to do is to write correct flow identifiers with correct directions, step numbers and/or classes, along with the commands in EMS programming language.

5. Conclusion

As a conclusion, we may say that the research questions find logical answers in the current paper. Our relatively nonstandard approach, which includes development of a command language with procedural features and the flow identifier operator, pays back. Complex queries can be written with very few lines of code, a polar coordinate system is employed in the flow identifier operator, human tasks are natively supported and easily extended with our IDE. Moreover, the EMS Script Editor enables the developer to avoid semantic errors, with the built in intelligent mechanisms of early discovery of semantic errors. The integration of EMS Script Editor with the Steps and Classes metadata, as well as the metadata structure itself, plays an important role in the early semantic errors discovery mechanism and the automatic GUI generation feature.

The future work includes the creation of intelligent generators of project prototypes, based on description of field domain and description of goals. The process of generations of steps and classes for the project can be highly automated. If the developer provides goal descriptions, some mechanisms described in [8] can be employed in order to generate the workflows as well.

The command interpreter will be optimized by including a phase, which indexes the commands, so that the line loop procedure can jump directly to the desired lines during execution, based on the index information of commands in the current trigger/function.

References

- [1] Cugnon de Sevicourt, O., & Tariel, V. (2011). Cameleon language part 1: Processor. *arXiv preprint arXiv:1110.4802*.
- [2] Dextro Research Ltd. (2009), IDE, Retrieved from: <http://dextro-research.eu/integrated-development-environment.html>
- [3] Dextro Research Ltd. (2009), Workflow Identifier, Retrieved from: <http://dextro-research.eu/workflow-identifier.html>
- [4] Ghaffari, B. (2000). *U.S. Patent No. 6,145,017*. Washington, DC: U.S. Patent and Trademark Office.
- [5] Grune Dick.(1999). *Parsing Techniques: A Practical Guide*, US: Springer.
- [6] Taverna Apache, Introduction to workflow management systems "What is a workflow management system?", Retrieved from: <http://www.taverna.org.uk>
- [7] Hinz, S., Schmidt, K., & Stahl, C. (2005, September). Transforming BPEL to Petri nets. In *International conference on business process management* (pp. 220-235). Springer, Berlin, Heidelberg.
- [8] Pashev, G., Totkov, G., Kostadinova, H., & Indzhov, H. (2016, June). Personalized Educational Paths through Self-Modifying Learning Objects. In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*(pp. 437-444). ACM.
- [9] Moran, T. P. (1981). The command language grammar: A representation for the user interface of interactive computer systems. *International journal of man-machine studies*, 15(1), 3-50.
- [10] Van Der Aalst, W. M., & Ter Hofstede, A. H. (2005). YAWL: yet another workflow language. *Information systems*, 30(4), 245-275.
- [11] Van Der Aalst, W. M., Ter Hofstede, A. H., Kiepuszewski, B., & Barros, A. P. (2003). Workflow patterns. *Distributed and parallel databases*, 14(1), 5-51.