

Implementation of Turing machines with the Scufl data-flow language

Tristan Glatard, Johan Montagnat

▶ **To cite this version:**

Tristan Glatard, Johan Montagnat. Implementation of Turing machines with the Scufl data-flow language. Workshop on Workflow Systems in e-Science, May 2008, Lyon, France. pp.1-10. hal-00683113

HAL Id: hal-00683113
https://hal.science/hal-00683113

Submitted on 27 Mar 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementation of Turing machines with the Scufl data-flow language

Tristan Glatard[1,2]*, Johan Montagnat[1]

1. CNRS / UNSA, I3S laboratory, Rainbow team
2. INRIA Sophia-Antipolis, Asclepios team

glatard@i3s.unice.fr ; johan@i3s.unice.fr

Abstract

In this paper, the expressiveness of the simple Scufl data-flow language is studied by showing how it can be used to implement Turing machines. To do that, several non trivial Scufl patterns such as self-looping or sub-workflows are required and we precisely explicit them. The main result of this work is to show how a complex workflow can be implemented using a very simple data-flow language. Beyond that, it shows that Scufl is a Turing complete language, given some restrictions that we discuss.

1. Introduction

Workflows are increasingly used by scientists in the field of e-Science as a way to set up in-silico experiments [6]. They are used as a glue to compose independent services, allowing their seamless integration in an application. Many different workflow models and languages have been proposed. In particular, data-flow and control-flow languages are classically distinguished. The former is said to be principally used in scientific applications, where the logic of the application is mainly driven by data dependencies, whereas the latter is more suited to business applications, which are likely to exhibit more complicated patterns, requiring sophisticated control operators.

Scufl is a simple data flow language which has been defined inside the myGrid UK eScience project and is used through the Taverna workflow manager by a large community of bioinformaticians [5]. In the context of grid applications running on high performance computing platforms, we developed MOTEUR, an optimized engine that interprets the Scufl language [1].

Because of its simplicity, it is often said that this language is not able to describe complex workflows and that it is bound to be used exclusively for simple pipeline applications. The goal of this paper is to study the expressiveness of the Scufl language, in order to determine whether it has theoretical limitations for the description of applications and if a shift to a more complete language has to be planned. Methods to study the expressiveness of a workflow language include workflow patterns [8] (the ability of the language to describe a set of pre-defined patterns is studied), schema relations [4] (the XML schema of the languages are semantically compared) and the study of the Turing completeness [2], which is the method we investigate here. Section 2 introduces the Scufl language and Section 3 overviews Turing machines. We then present the global principle of implementing a Turing machine using a Scufl workflow (Section 4). We show how it could be possible to expand it to a universal Turing machine workflow (Section 5) before discussing the remaining limitations of Scufl.

2. The Scufl language

Scufl is a data flow-oriented language that basically describes the pipeline of an application. Processors, sources and sinks of the workflow are first described and then chained using data links. A simple control link representing a coordination constraint without data transfer can also be used. Data composition operators allow to define iteration strategies between the input ports of a processor. Fig. 2 presents a summary of the graphical notations used to represent Scufl workflows.

Processors: Processors are actors of Scufl workflows. A processor is typically a Web Service embedding some user code but many other kinds of processors are defined in Scufl. For instance, string constants fire only once and return a single string value and Beanshells processors[1] embed a piece of Java code. No control constructs are available in Scufl. Yet, the

[1] http://www.beanshell

* Now affiliated to the University of Amsterdam

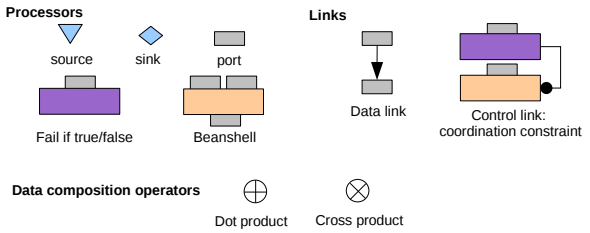


Figure 1. Graphical notations for Scufl

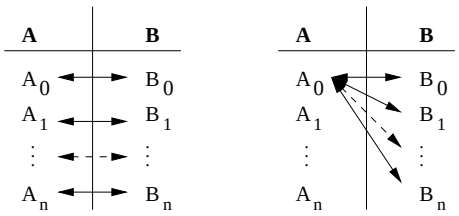


Figure 2. Dot (left) and cross (right) product.

`FailIfFalse` and `FailIfTrue` processors are special actors of the language defined to implement conditional branching in a workflow. Those processors fail or succeed depending on their Boolean input value, thus discarding or enabling the processors depending on them in the workflow. Sources and sinks correspond to the inputs and outputs of the workflow. Each of them may contain several data segments on which the workflow is iterated. Their content is not specified inside the Scufl document: it is independent from the workflow description and it is only known at runtime. The execution of a Scufl workflow is data driven: at start-up, sources deliver data to processors they are connected to through data links. These processors process their inputs and in turn pipe the processed data further into the workflow until sinks are reached.

Data composition operators: Attached to each processor with at least 2 input ports is an iteration strategy. Iteration strategies are used to control how multiple data items arriving in the input ports of a processor are combined to cause multiple invocations of the processor code. Two binary data composition operators may be used to compose an iteration strategy: the *dot* and *cross* products. Fig. 2 pictures the behavior of those operators. Given two input ports containing n data items, the dot product combines the items pairwise, leading to n invocations of the processor. Conversely, the cross product combines all the items of the first port with all the ones of the second port, leading to n^2 processor firings. If a processor has more than 2 input ports, cross and dot products have to be combined to define the whole iteration strategy. Brackets have to be properly set to define arithmetic priorities between operators.

Data and control links: Processors have input and output *ports* that can contain several data items and are connected to other ones with *data links*. A data link is just a pipe between an output port of a processor and an input port of another one. An output port can be connected to several input ports. In this case, the data items are broadcast to all the connected input ports. Similarly, several output ports can be linked to a single input port. In this case, data items are buffered into the input port according to their order of arrival. *Coordination constraints* can be specified in Scufl and provide elementary control links. Such a link specifies that a processor has to wait for another one before starting its execution, even if there are no data dependency between them.

Looping in Scufl: No control constructs such as *for* or *while* are available. Apart from the basic control link, the workflow is completely driven by the presence or absence of data in the input ports of a processor: a processor will fire if and only if all of its input ports contain adequate data. Moreover, a severe limitation of Scufl is that it is not possible to define variables nor evaluate expressions contrarily to more elaborated languages like BPEL [7]. In particular, global variables are not available and it may thus be difficult to maintain the state of a process. A consequence of the absence of variables is the absence of expressions and operators. Yet, implementing a loop is still possible in a data flow oriented way.

3. Turing machines

This Section provides a minimal and informal description of Turing machines. A complete presentation is provided *e.g* in [2]. A Turing machine is made of a tape, a head, a state and a transition function. The tape contains cells where symbols belonging to a finite alphabet are printed. The set of states is finite too. Particular states of the machine are the initial one and the set of final ones. The head is positioned on a given cell of the tape. At each iteration: (1) the head reads the current symbol on the tape ; (2) he transition function produces a new state, a new symbol and a head shift from the current state and symbol ; (3) the head writes the new symbol on the tape and moves one cell left or right depending on the shift given by the tran-

sition function and (4) the state of the machine is updated with the new state. If the new state is final, then the machine halts. Else, a new iteration starts.

Although this machine is very simple, the Church-Turing hypothesis states that every computable function can be computed by a Turing machine. Therefore, every language that is as expressive as a Turing machine is said to be Turing complete and would be able to implement any algorithm. A direct way to show that a language is Turing complete is to implement a Turing machine with it, as done for instance in [9] to show that C++ templates are Turing complete.

Following this line, the remainder of the paper details an implementation of a Turing machine in Scufl. We make the restrictive hypothesis that the tape of the implemented Turing machine is finite, thus preventing the algorithm to use an unbounded amount of resources.

4. A Turing machine in Scufl

4.1. Data flow description

Fig. 3 presents the implementation of a Turing machine in Scufl. The three sources `Ribbon`, `initState` and `stopState` respectively contain the input tape of the Turing machine, the initial state of the machine and the final states.

The `ReadInitSymbol` processor is used to initialize the machine with the first symbol to be interpreted. It has two parameters, the tape and the initial index. It simply extracts the corresponding character of the string obtained from the tape (`symbol=ribbon[index]`). The following symbols will be read by the `readSymbol` processor whose enactment is conditioned by the failure of the halting test. The obtained symbol is piped to the `Transition` processor which combines it with the current state of the machine to produce a new state (`outState`), a new symbol (`outSymbol`) and a movement of the head. The new state produced is looped back to the input of the `Transition` processor. Self-looping allows the `Transition` processor to maintain the state of the Turing machine. Remember that the use of global variables is not possible in Scufl. At a given iteration of the machine, the current state is obtained by proper data composition on the inputs of the `Transition` processor. This processor is the core of the Turing machine, as it implements the transition rules. In this Section, we assume that it is implemented with a Beanshell processor, which capture the whole logic with a piece of Java code. We will present a detailed Scufl implementation of this processor in the next Section.

The `movement` output of the `Transition` processor is passed to the `moveHead` processor. This processor only computes a new value of the head index from the shift passed by `Transition` and the current index value (`newIndex=index+shift`). Here again, the current value of the `index` variable is maintained thanks to self-looping: the output of the `moveHead` processor is connected to its `index` input. The `zero` string constant is also connected to the `index` input of `moveHead` to initialize the index value to 0.

The new index generated by `moveHead` is piped to the `write` processor as well. This processor also takes as input the `outSymbol` returned by `Transition` and the current tape of the machine. It replaces the character located at `index` on the current tape by `outSymbol` and returns the obtained new tape (`newRibbon=ribbon; newRibbon[index]=symbol;`). The state of the current tape is kept thanks to a self-looping.

`Transition` also returns the new state of the machine (`outState` parameter) which is passed to the `testHalt` processor. `testHalt` compares it to the final states provided as input of the workflow and returns a Boolean string piped to the `Fail_if_true` conditional processor (`bool=(stopState==state)`). Thus, if the current state of the machine corresponds to a final state, then the conditional processor fails and `readSymbol` does not fire, which makes the whole workflow stop because of the lack of symbols to consume. Else, `readSymbol` reads the next symbol and the machine iterates once again.

Finally, the `result` output of the workflow contains a history of the values of the tape, the last one being the result of the Turing machine.

The data composition operators of all the processors of the workflow except `testHalt` are dot products. Indeed, the processors have to correctly match the current values of the tape, head index and/or symbol over the successive iterations and it has to be done with a dot product. The `testHalt` processor has to test the value of the current state of the machine with *all* the final states, which justifies the presence of a cross product between its input ports.

4.2. Example on string length computation

The above-described workflow was implemented inside the Taverna workbench and executed with MOTEUR. It was tested on examples described with the Turing Machine Markup Language (TMML[2]) which

[2] http://www.unidex.com/turing/index.htm, (c) 2001 Unidex, Inc.

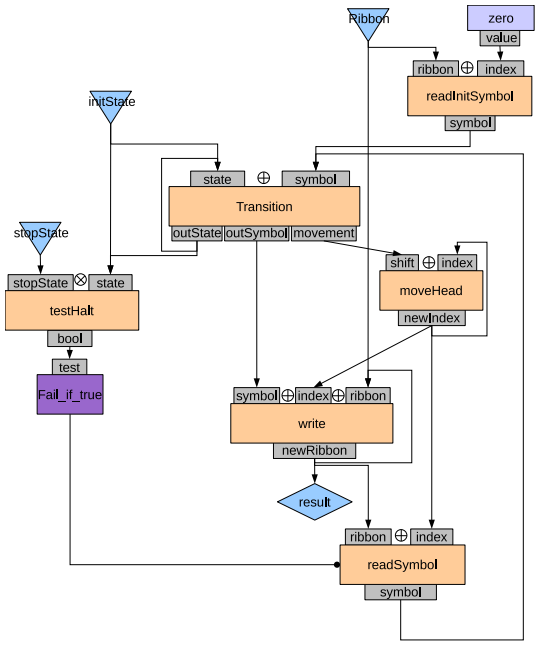


Figure 3. A Turing machine in Scufl

provides an easy-to-parse XML description of Turing machines.

In particular, a string length computation Turing machine was implemented. Its initial state is `start` and it has 2 final states, namely `string_is_null` and `stop`. 7 other states can be reached. At the end of the computation, the tape contains only an integer, which represents the length of the initial string.

The right of figure 4 displays this Turing machine executed with MOTEUR. The initial tape was the string `ab`. A total amount of 20 symbols have been read by the machine and passed to the `Transition` processor. The `testHalt` processor run 42 times. Indeed, including the initial state, 21 states have had to be tested by this processor and for each state to test, 2 invocations are required because there are 2 final states to compare with. The conditional processor only failed once (figured by red color), the last time it was invoked. The left of figure 4 shows the corresponding tape obtained for each iteration. The last one effectively only contains the length of the initial tape.

4.3. Limitations of this implementation

Parallelism exploitation: Scufl is intrinsically a parallel language: it allows processors to be iterated on several data sets. Given a suitable engine, data parallelism and pipelining can be exploited from the Scufl representation to obtain an efficient execution [1]. However, in our Turing machine implementation, enabling parallelism would completely puzzle the execution. For instance, if several different tapes are provided as input, the current state, index and tape of the machine could not be properly maintained. The use of a sub-workflow wrapping the Turing machine could help to cope with this problem.

Synchronization between conditional test and `readSymbol`: A more fundamental limitation of this Turing machine implementation is the synchronization between the conditional `Fail_if_true` processor and the `readSymbol` one. The firing of the `readSymbol` determines the firing of the `Transition` and subsequent processors. In Scufl, in absence of coordination constraints, the firing of a processor is determined by the availability of data items in its input ports. Because of the loops included in the workflow of our implementation of the Turing machine, data items are always available in the inputs ports of the `readSymbol` processor. Therefore, one should guarantee that the conditional processor is fired *before* each invocation of the `readSymbol` processor. Otherwise, the `readSymbol` processor could fire several times between two consecutive invocations of the `Fail_if_true` processor. This would certainly lead to some errors because of wrong stop condition detection. We solved this problem by firing the processors in a fixed order in our MOTEUR workflow engine, thus ensuring that the `Fail_if_true` processor is always fired before the `readSymbol` one. However, this kind of behavior is not specified in the Scufl document and is not handled by Taverna, the seminal Scufl enactor. A specification of the behavior of the engine should probably be included in the Scufl language, as it is done for instance in the MoML data-flow language through the definition of specific *directors* [3].

Code wrapping: As every workflow language, Scufl is able to define invocations to services, whose implementation is external to the workflow specification. Thus, one should keep in mind that some logic of the Turing machine implementation is embedded into those processors whose code is written using a traditional programming language such as Java. To properly assess the expressiveness of Scufl, one should be aware of that and limit the amount of non-Scufl code included inside the processors. In particular, the `Transition` processor includes a whole set of tests to implement the transition rules. Exaggeratedly, putting all the Turing machine logic inside a single processor would produce a correct implementation but would not prove anything about the expressiveness of the Scufl language. In the

1: ab
2: _ab
3: 0_ab
4: 0_ab
5: 0_ab
6: 0_ab
7: 0_ab_
8: 0_a__
9: 0_a__
10: 0_a__
11: 1_a__
12: 1_a__
13: 1_a__
14: 1_a__
15: 1____
16: 1____
17: 2____
18: 2____
19: 2____
20: 2____

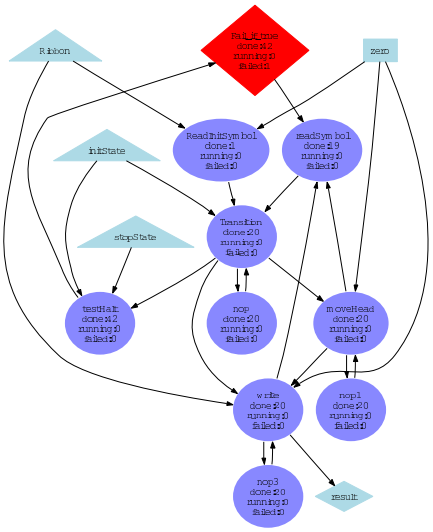


Figure 4. Right: Run of the Turing machine on a string length algorithm. Left: corresponding states of the tape for each iteration. Spaces are figured by '_'.

next section, we show how the `Transition` processor can be implemented in Scufl, using only Beanshells made of Boolean tests (`==`) and variable assignment (`=`). Thus, apart from those operators, the implementation of the Turing machine only requires the use of the array access operator `[]` (for the `readSymbol` and `write` processors) and of the incrementation (for the `moveHead` processor).

5. A universal Turing machine in Scufl

The above-described implementation of the Turing machine is not universal because the `Transition` processor has to be implemented for every set of rules. Moreover, as already suggested, it embeds a significant amount of code, which limits the evaluation of the expressiveness of the Scufl language. To cope with those limitations, we expanded the implementation of the `Transition` processor in Scufl.

The corresponding workflow of this processor is depicted on Fig. 5. It is made of a sub-workflow (`Nested_Workflow`) which tests the matching between a given transition rule and the current state and symbol of the machine. This sub-workflow has 7 different inputs. `currentState` and `currentSymbol` denote the current parameters of the machine. They must be compared to the conditions of the tested transition rule. Those sources respectively correspond to the `state` and `symbol` inputs of the `Transition` processor on Fig. 3. The 5 remaining inputs of the processor are new sources of the workflow. They define the rules of the Turing machine. `inState` and `inSymbol` are the conditions of the tested transition rule. `outSymbol`, `outState` and `movement` are the consequences of the transition rule. These are the value that must be returned by the sub-workflow if the tested transition rule matches the current parameters of the machine. The sub-workflow first tests the equality of the current parameters of the machine with the conditions of the tested transition rule. This is done through the `isEqual1` and `isEqual2` processors that just compare two strings and return a Boolean, which is tested by the conditional `Fail_if_false1` and `Fail_if_false2` processors. If *both* of the conditions are true, then the outputs of the rule are piped to the outputs of the sub-workflow through `nop` processors. Otherwise, the sub-workflow fails and does not return anything.

The `Nested_Workflow` sub-workflow is embedded into a global workflow. This is required (i) to allow to define iteration strategies between the different inputs of the sub-workflow even if they are not all connected to the same processor and (ii) to allow the `nop` processors to return only the correct output parameters of the transition rule. Indeed, if the sub-workflow was alone iterated on the whole transition rules, the `nop` processor would produce the complete set of `outState` parameters as soon as the `Fail_if_false1` processor would succeed. `nop` processors only correspond to a variable assignment (`output=input`) and `isEqual` processors implement a Boolean test (`a==b`). These are the only pieces of Java code required.

The iteration strategy of the `Nested_Workflow` is depicted on the bottom of Fig. 5. On the left side of the picture, the current state and symbol of the machine are composed with a dot product, to be able to associate only the right symbol with the right state. Similarly, on the right side of the picture, all the items of the transition rules are composed with dot products. The two terms in brackets are composed with a cross product, in order to test the current state and symbol with *all* the transition rules of the machine.

6. Conclusions

A universal Turing machine was implemented using the Scufl data-flow language. Clearly, this implementation would not have been possible without the use of data composition operators (cross and dot products) that introduce minimalist but sufficient control on the Scufl data-flow. Moreover, our implementation makes an intensive usage of loops, which are not prohibited

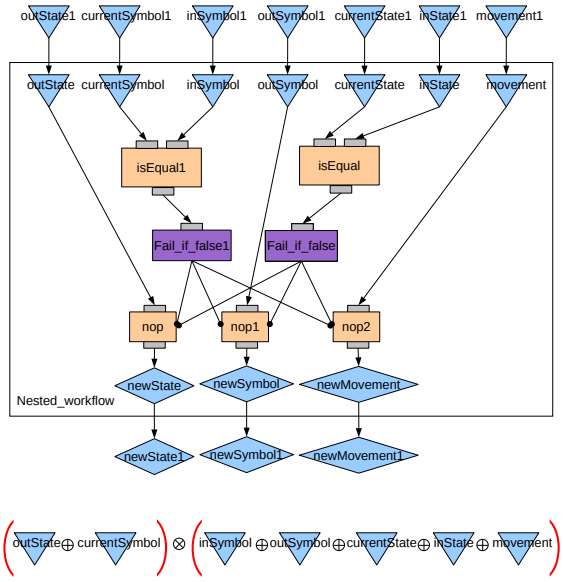


Figure 5. Top: the transition processor in Scufl. Bottom: iteration strategy of the `Nested_Workflow`.

by Scufl, as opposed to Direct Acyclic Graphs often used for scientific workflows. In particular, coupled with dot products iteration strategies, the use of self-looping enables a given processor to maintain a global variable in the workflow, whereas it is not possible to declare any variable in Scufl. Even if Scufl does not provide any conditional operator such as *if* or *switch*, the usage of the `Fail_if_true` and `Fail_if_false` processors provided by the language is sufficient for the implementation. The ability to define sub-workflows in Scufl documents is also required in our implementation. Sub-workflows allow a proper segmentation of the data sets and the use of data composition operators to define iteration strategies over the sources of a workflow, even if they are not connected to a single processor. This implementation includes pieces of Java code inside the Beanshell processors it uses. Nevertheless, those Beanshells only use a very restricted subset of Java. Only four operators are used: `=`, `==`, `[ ]` and `+`. These operators could easily be implemented as Scufl processors without loss of generality of the language.

Even if some restrictions remain, it is thus possible to conclude that Scufl is a Turing complete language. Therefore, it would theoretically be possible to implement any algorithm in Scufl. It highlights the fact that the expressiveness of such a data-flow oriented language is not as limited as expected, even if the language remains very simple and easy to manipulate. Moreover, a direct consequence of this result is that it is not possible to determine whether a Scufl workflow will complete or not.

Beyond those theoretical considerations, we presented an example of a development of a complex workflow exhibiting strong control requirements with a simple data flow oriented language. It exemplifies the use of specific patterns (in particular sub-workflows and self-looping) to solve expressiveness problems such as the definition of global variables. Thus, in spite of its simplicity, Scufl seems to be sufficiently expressive to describe complex applications. Adopting more complex languages including a whole set of control construct does not seem to be mandatory.

However, being able to implement a universal Turing machine in Scufl does not imply the ability to implement every algorithm in a user-friendly way. Implementing the workflow patterns described in [8] in Scufl would thus be an interesting perspective to this work.

7. Acknowledgments

We thank the Taverna developers for their development and maintenance. This work is funded by the French research program “ACI-Masse de données” (AGIR project) and the French ANR GWENDIA project (ANR-06-MDCA-009).

References

[1] T. Glatard, J. Montagnat, and X. Pennec. Efficient services composition for grid-enabled data-intensive applications. In *HPDC'06*, pages 333–334, Paris, June 2006.

[2] H. Lewis and C. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[3] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Conc. and Comp.: Practice & Experience*, 2005.

[4] J. Mendling and M. Müller. A Comparison of BPML and BPEL4WS. In *1st Conference Berliner XML-Tage*, pages 305–316, Berlin, 2003.

[5] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20):3045–3054, 2004.

[6] I. Taylor, E. Deelman, D. Gannon, and M. Shields. *Workflows for e-Science*. Springer-Verlag, 2007.

[7] BPEL4WS V1.1 specification, 2003.

[8] W. M. van der Aalst, A. H. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, July 2003.

[9] T. Veldhuizen. C++ Templates are Turing Complete. Technical report, Indiana University, 2003.