Turing equivalence of the *Ubiware Agent*

Michael Cochez

April 6, 2010

Abstract

In this paper, I'm proving that the *Ubiware Agent* (restricted in the use of RAB's) is Turing equivalent to a Turing machine. To prove this, I've to prove that a Turing machine can simulate a *Ubiware Agent* and that a *Ubiware Agent* is able to simulate a Turing Machine. The *Ubiware Agent* is defined in the Java language which is ran in a Java VM (which is known to be Turing equivalent). This implies that only one direction remains to be proven, that a *Ubiware Agent* is able to simulate a Turing Machine. I'll prove this by designing A *Ubiware Agent* beliefs structure in Semantic Agent Programming Language (*S-APL*) that is able to simulate the Turing Machine.

Contents

1 Introduction

1.1 What is *S-APL*?

S-APL is a language used for *Ubiware Agent* programming and can be summarized as in [1]

> Thus, an S-APL document is basically a statement of some agent's current or expected (by an organization) beliefs. S-APL is based on Notation3 (N3) and utilizes the syntax for rules very similar to that of N3Logic. N3 was proposed as a more compact, better readable and more expressive alternative to the dominant notation for RDF, which is RDF/XML. One special feature of N3 is the concept of formula that allows RDF graphs to be quoted within RDF graphs, e.g. {:room1 :hasTemperature 25} :measuredBy :sensor1. An important convention is that a statement inside a formula is not considered as asserted, i.e., as a general truth. In a sense, it is a truth inside a context defined by the statement about the formula and the outer formulas. In S-APL, we refer to formulae as context containers.

The *S-APL* syntax, as described in [3] is
The description of *S-APL* notation follows:

- A statement is a white-space-separated sequence of subject, predicate and object
- Dot (.) followed by a white space separates statements of the same level, i.e. S P O . S P O
- Semicolon (;) followed by a white space allows making several statements about the same subject, i.e. S P O ; P O
- Comma (,) followed by a white space allows making several statements having common subject and predicate, i.e. S P O , O
- { } denotes reification, it may appear as the subject or the object of a statement and has to include inside itself one or more other statements, e.g. S P { S P O } or { S P O } P { S P O }. Reification always implies a context; however, the relation is not necessarily 1-to-1. E.g. {S P O} P O ; P O implies that the statement in {} is linked to two different contexts defined as given.
- Colon (:) is used to specify an URI as a combination of the namespace and the local name, i.e. ns:localname There can be default namespace, the colon is used anyway, i.e. :localname.
- @prefix prefix: namespace links a prefix to a namespace.
- URIs given directly are to be inside < >, i.e. <http://someaddress>.
- Literals containing whitespaces, {, }, <, > , ", or : are to be inside " ", i.e. "some literal" .
- Comments are java-style, i.e. /* comment */ as well as // comment <end of line>
- Character escaping is java-style as well, i.e. e.g. for " symbol, use \ " while for backslash symbol itself, use \\
- N3 syntax for anonymous nodes with [ ], i.e. S P [P O] or [P O] P O is also supported.
- N3 syntax for RDF lists of resources with ( ) , i.e. (R R R ..) P O or S P (R R R ..) is also supported.

For the remainder of this document I assume using the namespace sapl: as

`<http://www.ubiware.jyu.fi/sapl#> .`

G is considered the main *Ubiware Agent* belief container and contains everything the *Ubiware Agent* assumes to be general truth. I consider *Ubiware Agent* without the possibility to start external RAB's (Reusable Atomic Behavior's are pieces of java code which are executable by *Ubiware Agent* to support their interaction to the outside world) For the proof I need a subset of the *S-APL* language constructs. Those will be explained in the following sections, for these and other constructs see [3].

Unconditional commitment to adding a belief

If the *Ubiware Agent* has a belief: "sapl:I sapl:add {X}" in G , then the *Ubiware Agent* copies "X" to G and erases the unconditional commitment.

Unconditional commitment to removing a belief

If the *Ubiware Agent* has the belief: "sapl:I sapl:remove {X}" in G , then all believes matching to "X" are removed from G (For matching rules see [3])

Conditional commitment

If the *Ubiware Agent* has a belief: "{X_1 . X_2 . ... . X_n} => {X}" in G , where $n \in \aleph \setminus \{0\}$ then "X" is added to G if all of "X_1, X_2, ... , X_n" can be matched against G . (Possibly linking values to variables denoted by ?variablename.) After applying the conditional commitment, it is removed from G. However, if a conditional commitment appears inside the subject container of {} sapl:is sapl:Rule then the rule gets executed but doesn't get removed.

1.2 What is a Turing Machine and Turing equivalence?

1.2.1 Turing Machine

A Turing Machine is a theoretical model for computation and is formally defined in [2] as a seven tuple

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle \text{ where}$$

- Q is the finite set of *states*.
- $\Sigma \subseteq \Gamma \setminus \{B\}$ is the set of *input symbols*.
- Γ is a finite set of the *tape symbols*.
- $\delta : Q \setminus F \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is a partial function called the *transition function*.

  If $\delta : (q, X) \rightarrow (p, Y, D)$ then p is the next state of the Turing Machine, Y is the replacement symbol written in the cell being scanned. And D is the direction in which the head of the Turing Machine will be moved.
- $q_0 \in Q$ is the *initial state* .

- $B \in \Gamma \setminus \Sigma$ is the *blank symbol* The blank symbol appears initially in all but the finite number of initial cells that hold the input symbols.
- $F \subseteq Q$ is the set of *final* or *accepting states*.

Furthermore, the Turing Machine has an infinite tape of cells, initially containing the input to the machine and each non-input containing cell contains B. I assume the tape to be infinite in one direction only. This doesn't decrease the computational power of the machine [2]. The Turing Machine has a head, which initially scans the cell containing the first input symbol. A step of the machine means the application of δ. This means, read the state q of the machine and the symbol on the currently scanned cell and apply δ; write Y in the scanned cell, change the state of the machine to p and move the head in direction D.

1.2.2 Turing equivalence

I'm using the following definition for Turing equivalence:

> A machine V is Turing equivalent with a machine W $\Leftrightarrow$ V can simulate the functionality of W and vice versa.

Concrete I'm going to give a proof that the *Ubiware Agent* is Turing equivalent with the previously defined Turing Machine. The proof that the *Ubiware Agent* is simulatable by the Turing Machine is given by the fact that the reference implementation is made in Java (definition by construction). [4] Thus only one direction remains to be proven: The *Ubiware Agent* is able to simulate the Turing Machine.

1.3 Why proof Turing equivalence?

A Turing Machine is a model for all known computing machines. Knowing that a newly built machine (in this case the *Ubiware Agent*) is Turing equivalent is sufficient as a proof that it is able to perform all possible algorithms.

2 Proof by simulation

2.1 Building the machine

I proof the fact that *Ubiware Agent* is able to simulate the Turing Machine by writing *S-APL* code which, when injected in G , is able to simulate the working of the Turing Machine. The transformation from the Turing Machine definition to the *S-APL* code is as follows: (Note that we use a method which simulates the "Instantaneous descriptions for Turing Machine" described in A)

I introduce the following notation:

X $\in F$ with F a *S-APL* container and X a valid *S-APL* belief $\Leftrightarrow$ X is part of the container F.

Let G be the initially empty container of the *Ubiware Agent* beliefs.

Defining the namespaces

For the simulation, we need to define the namespaces of the used elements:

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@prefix tape: <http://users.jyu.fi/miselico/seminaari/tape#> .
@prefix alf: <http://users.jyu.fi/miselico/seminaari/alphabet#> .
@prefix machine: <http://users.jyu.fi/miselico/seminaari/machine#>.
@prefix state: <http://users.jyu.fi/miselico/seminaari/state#> .
@prefix rule: <http://users.jyu.fi/miselico/seminaari/rule#> .
@prefix dir: <http://users.jyu.fi/miselico/seminaari/direction#>
```

$\in G$

Machine description

Let : $myTuringMachine$, : $myTuringTape$, : $myTuringHead$ be unique names for this Turing Machine, its tape and its head. First we define a general Turing Machine:

```
:myTuringMachine machine:hasTape :myTuringTape .
:myTuringMachine machine:hasHead :myTuringHead .
:myTuringMachine machine:initialSate q_0 .
:myTuringMachine machine:hasBlankSymbol B
```

$\in G$

Machine initialization

Then we define a rule which must give the Turing Machine its initial position:

```
{
        ?theTuringMachine machine:hasTape ?tape .
        ?tape tape:hasCell ?cell .
        ?cell tape:hasSequenceNr 0 .
        ?theTuringMachine machine:hasHead ?head .
        ?theTuringMachine machine:hasInitialState ?initialstate
}
=>
{
        ?head machine:hasPosition ?cell .
        ?theTuringMachine machine:hasState  ?initialState
}
```

$\in G$

Transformation of δ

Now we model the rules of the Turing Machine:
Let every member of the transition function have a unique name, I denote it with :rule, but it is different for each element.
For rules which make the head move to the right:
$\forall((state : validState, alf : OChar), (state : resultState, alf : RChar, R)) \in \delta$:

```
:myTuringMachine machine:hasRule :rule
```

$\in G$ and

```
:rule rule:hasOriginalSymbol alf:OChar ;
      rule:hasReplacementSymbol alf:RChar ;
      rule:hasDirection dir:right ;
      rule:validInState state:validState ;
      rule:transitionToState state:resultSate .
```

$\in G$

For rules which make the machine move to the left:
$\forall((state : validState, alf : OChar), (state : resultState, alf : RChar, L)) \in \delta :$

```
:myTuringMachine machine:hasRule :rule
```

$\in G$ and

```
:rule rule:hasOriginalSymbol alf:OChar ;
      rule:hasReplacementSymbol alf:RChar ;
      rule:hasDirection dir:left ;
      rule:validInState state:validState ;
      transitionToState state:resultSate .
```

$\in G$

Later we will refer to these rules in the beliefs of *Ubiware Agent* which originate from the definition of th Turing Machine as *"TMRules"*.

TMSimulationRule

Then we define a rule (TMSimulationRule) which simulates the working of the Turing Machine. I've placed the TMSimulationRule to the appendix B because it is quite spacious. It takes the original state of the machine and the symbol the head is currently pointing at and uses a TMRule to calculate the new position of the head, the symbol which should het written in the scanned cell and the new state. The right hand-side of the TMSimulationRule, replaces the scanned cells content by the new symbol and changes the machine state. Then a new rule is added to check whether the cell which should be scanned next is already existing in the *Ubiware Agent* belief, if not, it gets created with the blank value. Then a rule is added which moves the head to the cell on the right or left (depending of the direction of the TMrule) of the scanned cell as soon as it exists.

2.2 Modeling the input for the machine

The input which is put to the tape of the Turing Machine , must be modelled for the *Ubiware Agent*, we do that as follows:

Let $w = s_1 s_2 ... s_n$ be the input to the Turing Machine, then $\forall s_k \in w :$

```
:myTuringTape tape:hasCell :cell_k .
:cell_k tape:hasSequenceNr k;tape:hasValue s_k
```

$\in G$

2.3 Proof of equivalence by induction

I proof that the simulation in the *Ubiware Agent* is a simulation of the original Turing Machine by showing per induction that the Instantaneous description after every step of the Turing Machine is corresponding to the state of the *Ubiware Agent* after execution of TMSimulationRule. Assume a Turing Machine defined as in 1.2.1 and input string $w = s_1 s_2 ... s_n$

Basic case

Initially, the Instantaneous description of the Turing Machine is $ps_1 s_2 ... s_n$. The corresponding initial part of G is:

```
:myTuringHead machine:hasPosition cell_1 .
:myTuringMachine machine:hasState p_0
```

and the tape looks as follows:

```
:myTuringTape tape:hasCell cell_1 , cell_2 , cell_3 , ... , cell_n
cell_1 tape:hasSequenceNr 1; tape:hasValue s_1 .
cell_2 tape:hasSequenceNr 2; tape:hasValue s_2 .


cell_n tape:hasSequenceNr n; tape:hasValue s_n
```

Inductive step for rules which move the head to the left.

After a number of steps the instantaneous description of the Turing Machine is $X_1X_2...X_{i-1}qX_iX_{i+1}..X_n$

If then

$\delta\{q, X_i\} \rightarrow \{p, Y, L\}$, the next ID will be $X_1X_2...X_{i-2}pX_{i-1}YX_{i+1}..X_n$

In the beliefs of the *Ubiware Agent*, the same will happen : the initial Instantaneous description looks like :

```
:myTuringHead machine:hasPosition cell_i .
:myTuringMachine machine:hasState q
```

and the tape looks as follows:

```
:myTuringTape tape:hasCell cell_1 , cell_2 , cell_3 , ... , cell_n
cell_1 tape:hasSequenceNr 1; tape:hasValue X_1 .
cell_2 tape:hasSequenceNr 2; tape:hasValue X_2 .
...
cell_{i-1} tape:hasSequenceNr i-1; tape:hasValue X_{i-1} .
cell_{i} tape:hasSequenceNr i; tape:hasValue X_{i} .
cell_{i+1} tape:hasSequenceNr i+1; tape:hasValue X_{i+1} .
...
cell_n tape:hasSequenceNr n; tape:hasValue s_n
```

The *Ubiware Agent* now applies the TMSimulationRule which maps the ?rule variable with a TMrule which maps the state q, and value X_i in the scanned cell to something. This rule must be of the form:

```
?? rule:hasOriginalSymbol X_i ;
   rule:hasReplacementSymbol ?? ;
   rule:hasDirection ?? ;
   rule:validInState q ;
   rule:transitionToState ?? .
```

The only rule which can have this form is a rule which originate from an element of δ of the form $((q, X_i), (??, ??, ??)) \in \delta$. The only element of δ which has this form is $((q, X_i), (p, Y, L))$ because δ is a function. Thus the only applicable TMrule is (by construction):

```
:rule rule:hasOriginalSymbol X_i ;
      rule:hasReplacementSymbol Y ;
      rule:hasDirection dir:left ;
      rule:validInState q ;
      rule:transitionToState p .
```

application of the TMSimulationRule with this TMrule, yields the following result:

```
:myTuringHead machine:hasPosition cell_{i-1} .
:myTuringMachine machine:hasState p
```

and the tape looks as follows:

```
:myTuringTape tape:hasCell cell_1 , cell_2 , cell_3 , ... , cell_n
cell_1 tape:hasSequenceNr 1; tape:hasValue X_1 .
cell_2 tape:hasSequenceNr 2; tape:hasValue X_2 .
...
cell_{i-1} tape:hasSequenceNr i-1; tape:hasValue X_{i-1} .
cell_{i} tape:hasSequenceNr i; tape:hasValue Y .
cell_{i+1} tape:hasSequenceNr i+1; tape:hasValue X_{i+1} .
...
cell_n tape:hasSequenceNr n; tape:hasValue s_n
```

which is equivalent to the ID of the simulated Turing Machine.

Inductive step for rules which move the head to the right.

The prove for right moves is equivalent to the one for left moves, I won't repeat the reasoning.

For right moves, there are two exceptional cases though[2]:

- If $i = n$ then the $i+1$st cell holds a blank, and that cell was not part of the previous ID. Thus, we instead have $X_1X_2...X_{n-1}qX_n \vdash X_1X_2...X_{n-1}YpB$ This case is covered by the construction of TMSimulationRule because if the cell which should be scanned next does not exist yet, it gets created with the blank value.
- If $i = 1$ and $Y = B$, then the symbol B written over X_1 joins the infinite sequence of leading blanks and does not appear in the next ID, Thus, $qX_1X_2...X_n \vdash pX_2...X_n$ Also this case is covered, the *Ubiware Agent* keeps the belief about the blank cell being existing, but that does not affect the state.

3 Limitations of the proof

The main limitation of this proof is that *Ubiware Agent* is used as a theoretical model. The concrete implementation of the *Ubiware Agent* has constraints which would limit its possibility to model the universal Turing Machine. Those limitations are mainly the limited size of the belief structure (it has to fit in the memory of the Java VM) and the limitation of integers to $2^{31} - 1$. (With some tricks this could be increased to $2 * 2^{31}$ but is still is not infinite. In theoretical computer science this limitation is often relaxed [2] by assuming that the computer can be offered an infinite amount of memory in its simulation or by stating that certain limits are "infinite enough".

4 Conclusions

I wanted to proof that the *Ubiware Agent* is able to simulate a Turing Machine as defined in [5]. First I needed to construct beliefs for the *Ubiware Agent* based on the description of the Turing Machine. Then I transformed the input to become suitable to feed to the *Ubiware Agent*. Then I showed by induction using Instantaneous descriptions as defined in [2] that *Ubiware Agent* is able to simulate the Turing Machine. Now, because the *Ubiware Agent* is simulated in

Java, and the *Ubiware Agent* is able to simulate the Turing Machine, we know that they are Turing Equivalent. The main limitation of the proof is that the *Ubiware Agent* is used as an agent with infinite storing capabilities, which it is not in a concrete implementation.

References

[1] Katasonov A. and Terziyan V. (2008) Semantic Agent Programming Language (S-APL): A Middleware Platform for the Semantic Web. In: Proc. 2nd IEEE International Conference on Semantic Computing (ICSC'08), August 4-7, 2008, Santa Clara, USA IEEE, pp.504-511

[2] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). Introduction to Automata Theory, Languages, and Computation (2nd ed.). Addison-Wesley.

[3] Semantic Agent Programming Language (*S-APL*) Developer's Guide by Arthem Katasonov (Jyväskylän Yliopisto)

http://users.jyu.fi/ akataso/SAPLguide.pdf on 6 april 2010.

[4] Website about *S-APL* and the reference implementation.

http://users.jyu.fi/ akataso/sapl.html on 6 april 2010.

[5] A. M. Turing,"Proc. London Math. Soc. 42", (230–265), "On Computable Numbers, with an application to the Entscheidungsproblem", 1936 volume 2

A Instantaneous description of a Turing Machine

The Instantaneous description of a Turing Machine is (according to [2] and [5]: A string $X_1X_2...X_{i-1}qX_iX_{i-1}...X_n$ in which:

1. q is the state of the Turing Machine
2. The tape head is scanning the ith symbol from the left.
3. $X_1X_2...X_n$ is the portion of the tape between the leftmost and the rightmost non-blank.

B The TMSimulationRule

```
@prefix sapl: <http://www.ubiware.jyu.fi/sapl#> .
@prefix tape: <htpp://users.jyu.fi/miselico/seminaari/tape#> .
@prefix alf: <http://users.jyu.fi/miselico/seminaari/alphabet#> .
@prefix machine: <http://users.jyu.fi/miselico/seminaari/machine#>.
@prefix state: <http://users.jyu.fi/miselico/seminaari/state#> .
@prefix rule: <http://users.jyu.fi/miselico/seminaari/rule#> .
@prefix direction: <http://users.jyu.fi/miselico/seminaari/direction#>

{
//turingMachine functional rule
{
  ?turingMachine machine:hasHead ?head .
  ?turingMachine machine:hasSate ?originalState .
  ?head machine:hasPosition ?originalPosition .
```

```
  ?originalPostion tape:hasSequenceNr ?originalPositionSequenceNr ;
                   tape:hasValue ?originalPosiotionValue .
  ?newPositionSequenceNr sapl:expression ?originalPositionSequenceNr-1 .
  ?turingMachine machine:hasRule ?rule .
  ?rule rule:hasOriginalSymbol ?originalPositionValue ;
        rule:hasReplacementSymbol ?originalPositionReplacementValue ;
        rule:hasDirection direction:left ;
        rule:validInState ?originalSate ;
        transitionToState ?newState .
}
=>
{
  //change the value in the scanned cell ?position
  sapl:I sapl:remove {?originalPosition tape:hasValue ?originalPositionValue} .
  sapl:I sapl:add {
           ?originalPosition tape:hasValue ?originalPositionReplacementValue
                  } .

  //change the state of the machine
  sapl:I remove {?turingMachine machine:hasState ?originalState} .
  sapl:I sapl:add {?turingMachine machine:hasSate ?newState} .

  //move the position of the head to the left
  sapl:I sapl:remove {?head machine:hasPosition ?originalPosition} .
  sapl:I sapl:add {
      //If the cell does not exist yet, we create it
      {sapl:I sapl:doNotBelieve {
                  ?turingMachine machine:hasTape ?tape .
                  ?tape tape:hasCell ?newPosition .
                  ?newPosition tape:hasSequenceNr ?newPositionSequenceNr
                                 } .
       ?turingMachine machine:hasBlankSymbol ?blank .
       sapl:Now sapl:is ?now .
       ?newCellIdentifier sapl:expression
                   "(?now+?turingMachine+Cell)+?newPositionSequenceNr"
       }
      ->
        {sapl:I sapl:add {
                  //generate a new cell with blank value
                 ?tape tape:hasCell  ?newCellIdentifier.
                 ?newCellIdentifier tape:hasSequenceNr ?newPosition;
                                    tape:hasValue ?blank
              }
          } .
          //move the position
        {?turingMachine machine:hasTape ?tape .
         ?tape tape:hasCell ?newPosition .
         ?newPosition tape:hasSequenceNr ?newPositionSequenceNr
        }=> {?head machine:hasPosition ?newPosition}
      }
   }

} sapl:is sapl:Rule
```