Computer Science • 15 (3) 2014 http://dx.doi.org/10.7494/csci.2014.15.3.293

Jarosław Rudy

TURING MACHINE APPROACH TO RUNTIME SOFTWARE ADAPTATION

Abstract *In this paper, the problem of applying changes to software at runtime is considered. The computability theory is used in order to develop a more general and programming-language-independent model of computation with support for runtime changes. Various types of runtime changes were defined in terms of computable functions and Turing machines. The properties of such functions and machines were used to prove that arbitrary runtime changes on Turing machines are impossible in general cases. A method of Turing machine decomposition into subtasks was presented and runtime changes were defined through transformations of the subtask graph. Requirements for the possible changes were considered with regard to the possibility of subtask execution during such changes. Finally, a runtime change model of computation was defined by extension of the Universal Turing Machine.*

Keywords runtime changes, dynamic modification, computability theory, Turing machines

1. Introduction

Models of computation are theoretical concepts that allow us to reason about the nature and limitations of computation and computability, forming a basis for all existing physical implementations of computing machines and modern computers. The theoretical concept of such models has received a great deal of attention since the 1930s. Many different models have been proposed since then, including the untyped lambda calculus developed by Church [2], μ-recursive functions by Kleene [4], and the *Turing machine* [8]. In fact, all three models were proven to be equivalent by Rosser [6]. While the first two models are purely mathematical, the Turing machine is closest to the description of a real machine, therefore able to directly affect the design of real-life computers.

The Turing machine inherently models static computation, as the algorithm is contained in the transition function (the state graph), which does not change. The Universal Turing Machine (UTM) is a generalization of this machine, described by Turing in his original paper [8]. The UTM simulates another Turing machine, whose definition is provided on the tape. Since the tape can change, the UTM has the potential for dynamic change of algorithms while the answer is computed. No sufficient theory or models of computation for dynamic computations exist, however.

The need for such a theoretical model is clearly evident from practice, as there are various reasons for change in software after it has been released. Those changes can be distinguished into several types based on their cause:

1. Requirement analysis fault – takes place when client is not completely aware of its own needs or when the communication between the client and the developers (programmers) is faulty.
2. Implementation fault – meaning that the created software does not meet its requested specification.
3. Requirement change – any change in functionality that was not planned by the client, but is necessary (*e.g.* due to new legal regulations).
4. Environment change – applies mostly to larger computer systems, and usually means the changes in available resources (number of network nodes, connection throughput *etc*).
5. Software optimization – includes a decrease in resources needed by the software (*e.g.* time or space complexity).

The above causes are often unpredictable as they employ a human factor and, thus, cannot be fully eliminated through the use of software engineering. Therefore, much effort has been made through the years to make applications modifiable and changeable to suit the various needs of the user. This is often achieved by a system of updates or plug-ins. This method, however, usually requires that the application in question is restarted after every update, greatly limiting the benefits and convenience of a dynamic change. Runtime change without shutting down or restarting the application is more desirable, but it is difficult to achieve.

Many papers exist concerning the research of *Runtime Software Adaptation* (RSA) of computer programs. Here, we will mention them briefly. Mukhija and Glinz [5] and Wang *et al.* [12] use the so-called *graph of components* model, where the software components of the system are represented as a graph. Runtime change is done by changing the components of the graph, which affects the target system. Valetto and Kaiser [9] use a more complex system in which a set of probes and gauges collects and processes data from the target system. Then a set of controllers makes adaptation decisions based on the model of software. In this case, the model can be represented by a set of rules and constraints that the target system needs to meet. Lastly, a set of effectors is used to change the target system.

It is important to note that approaches exist that provide Runtime Software Adaptation by the use of specific programming languages or paradigms. For example, Villazón and Binder [11] proposed an approach using aspect-oriented programming dedicated solely for Java. On the other side, Valetto *et al.* [10] rely on the software agents paradigm in their solution, once again for Java. Another approach was proposed by Rudy [7], which is a low-level tool aimed at C++ applications for Linux.

The above examples prove that there is much interest in runtime adaptation. However, existing approaches are mostly practical ones, and the lack of theoretical approaches and theorems is apparent. This is natural, considering that practical research often precedes the construction of corresponding theories. For example, in the field of operation research, many metahueristic algorithms were developed years before their theoretical properties (like proofs of convergence) were considered. Therefore, this paper will focus on the possibility to apply the concept of runtime changes on the level of models of computation, like the Turing machine, thus creating a dynamic model of computation. This will allow us to identify and study the properties and limitation of such a model and, by extension, to apply the concept of runtime change to any computer system that follows this model, regardless of the programming language or paradigm used.

The remainder of this paper is organized as follows: Section 2 introduces the concept of dynamic computable functions and uses them to prove that arbitrary runtime changes on arbitrary deterministic Turing machines are impossible; Section 3 describes a method of decomposition of Turing machines and computable functions - it also presents various types of runtime changes for such machines and their limitations; Section 4 proposes a dynamic model of computation and an algorithm used to perform the runtime changes; Finally, Section 5 contains the conclusions.

2. Dynamic computable functions

Many different models of computations were proposed, including Turing machines, λ-calculus and μ-recursive functions amongst others. Moreover, the three above-mentioned models have been proven to be equivalent by Rosser [6], and all other models are either equivalent or weaker than them in terms of computation power (*i.e.* the class of problems solvable with them) or not feasible to our current knowledge,

like Turing machine with oracle. By R we denote the set of all problems that can be solved by the mentioned models, meaning that for every $r \in R$ there exists a Turing machine, λ-function and μ-recursive function that solves r. Elements from R can be treated as functions in the form of $f : \mathbb{N} \to \mathbb{N}$. Thus, we define a set of *computable functions* (in terms of the given model) denoted as CF.

It is currently not known whether all functions "intuitively treated as effectively[1] calculable" are computable by the Turing machine. The Church-Turing thesis states that it is so, which is supported by the equivalence of mentioned models; but since the thesis itself uses vague concept of "effective calculability" it cannot be mathematically proven. Fortunately, our aim is to model computations done by real-life computers, therefore the restriction to Turing-computable functions is adequate.

We will now use the concept of computable functions to model causes for software runtime changes described in the previous chapter. Each piece of software is created to fulfill some sort of client requirements, which in turn generates revenue for the software company (*i.e.* developers), and those requirements can be treated as function f that assigns expected system output y of the system for every input x:

$$f \colon X \nrightarrow X, \qquad f(x) = y \tag{1}$$

By X, we mean all possible states of the environment of the software (*i.e.* memory). Let us also note that f is a partial function in general. Assuming the binary nature of the computers, we can treat each memory cell as a single binary digit and the entire memory as a single natural number *i.e.* $X = \mathbb{N}$. Therefore, the client's requirements are a computable function. Let us consider the following computable functions:

- $r(x)$ – requirements intended (needed) by the client,
- $c(x)$ – requirements communicated from the client to the developers,
- $r'(x)$ – new (changed) requirements,
- $p_A(x)$ – requirements p carried out by Turing machine A.

Then, the runtime change causes from before can be formally defined as follows:

$$\text{Cause 1:} \quad r(x) \neq c(x) \tag{2a}$$

$$\text{Cause 2:} \quad p_A(x) \neq c(x) \vee p_A(x) \neq r(x) \tag{2b}$$

$$\text{Cause 3:} \quad r'(x) \neq r(x) \tag{2c}$$

$$\text{Causes 4 and 5:} \quad comp(p_B(x)) \in O(comp(p_A(x))) \tag{2d}$$

The cases 2a through 2c are self-explanatory; however, case 2d requires further elaboration. In general, the last case means that the complexity of the algorithm (Turing machine) B is at most the complexity of the algorithm A for some computable function p. Therefore, asymptotically B is never worse than A in terms of some complexity measure. Typical measures include time and space complexity. The

[1] effectively, but not necessarily efficiently

proposed form of the case 2d is true yet impractical, as it is still possible for A and B to have the same complexity, at which point replacing A with B is baseless. In practice, we want to perform the change when $comp(B) < comp(A)$ for every x or when $comp(B) \in o\ (comp(A))$.

Before we will look closer at the changes of computable functions, let us formalize the one-tape deterministic Turing machine as our basic model of computation. The deterministic Turing machine (DTM) is defined (after Hopcroft and Ullman [3]) as 7-tuple $M =< Q, \Gamma, b, \Sigma, \delta, q_0, F >$ as follows:

- Q – finite, non-empty set of internal states,
- $F \subset Q$ – finite, non-empty set of final states,
- $q_0 \in Q$ – initial state,
- Γ – finite, non-empty set of tape symbols (tape alphabet),
- $b \in \Gamma$ – blank symbol,
- $\Sigma \subseteq \Gamma$ – input alphabet,
- $\delta : Q \setminus F \times \Gamma \to Q \times F \times \{L, R\}$ – the transition function.

The distinction is made between the alphabet used for input (Σ) and the alphabet used for both calculations and output (Γ). At the beginning, the blank symbol b occupies all spaces on the infinite tape (in both directions), except for the spaces containing the input. Moreover, the blank symbol is the only one that is allowed to infinitely occupy many spaces. δ is a partial function that, for any non-final internal state and any tape symbol, produces the symbol to write to the tape, the next internal state to transit into, and L or R indicating whether we should move the head of the machine (or the tape) by one space to the left or right. When the machine enters one of the final states, it halts. For simplicity, we can assume that $\Sigma = \Gamma$ and that only one final state exists, called HALT. Moreover, since Turing machines compute computable functions, it means that the natural numbers can be expressed as words from the alphabet Γ.

2.1. Arbitrary changes

In general, we want to be able to perform arbitrary runtime changes. Let us assume that the client's requirements are in the form of some arbitrary computable function $f(x)$. Let M_f be a DTM created by developers such that M computes $f(x)$, but is otherwise arbitrary. Then, user requirements change to the form represented by computable function $g(x)$, such that $g(x) \neq f(x)$. In other words, we want to replace M_f with arbitrary M_g, while M_f may not have halted yet. We assume that the only thing known to M_g is the state of the tape left by M_f when it is replaced. Let us define a *partial output* of a DTM.

Definition 1. *Let M be a deterministic Turing machine.* Partial output *of M is a function $z_M(x, n) : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N}$ that yields the state of the tape (a non-negative integer) after n steps of M for input x.*

Partial output is a function of two natural numbers; but since we can uniquely encode two natural numbers into a single one by the use of the pairing functions, we can therefore treat $z_M(x, n)$ as $\mathbb{N} \nrightarrow \mathbb{N}$ whenever needed. Next, let us consider the properties of the partial output function. In particular, such a function should be a one-to-one function (an injection) as to prevent ambiguity; otherwise, we wouldn't be able to determine the original value of x based on the known values of n and z_M. For $z_M(x, n)$ to be one-to-one function, the equation $z_M(x_1, n_1) \neq z_M(x_2, n_2)$ must hold for every $x_1, x_2 \neq x_1$ and $n_1, n_2 \neq n_1$.

The above function relates to the problem of reversibility of computation. Bennett [1] showed that, for specific Turing machines, such a reversibility is possible, as a Turing machine can keep intermediate results of every operation, giving the possibility to backtrack. This, however, is not a general solution, and it is quite expensive in terms of both time and space complexity (requiring additional steps and tape space to store results and backtrack). In our case, we want to avoid such complexity; thus, we will focus on the properties of the partial output function $z_M(x, n)$ for more general deterministic Turing machines.

Let us consider deterministic Turing machine M such that $\Gamma = \Sigma = \{\mathbf{1}, \mathbf{0}\}$. Moreover, let M work as follows: given a sequence of **1**s M travels along the tape until either the number of scanned **1**s exceeds 10 or **0** is found, whichever comes first. If the number of **1**s exceeds 10, M travels back along the tape replacing all **1**s with **0**s. If **0** is found first, M clears the tape as well and then prints a single **1**. M outputs 1 when $x \leqslant 10$ and 0 otherwise. Such DTM computes function f:

$$f(x) = \begin{cases} 1 & \text{if } x \leqslant 10, \\ 0 & \text{if } x > 10. \end{cases}$$

Let us consider $f(8)$. At the beginning, the number of **1**s on the tape equals 8. M clears the tape at some point; thus, the tape contains zero **1**s. Since M can clear only one **1** per step, it follows that after some number of steps n_1 the number of **1**s on tape must equal 4, so $z_M(8, n_1) = 4$. The same goes for $f(12)$: after the n_2 steps, the number of **1**s will equal 4, so $z_M(12, n_2) = 4$. Therefore, $z_M(8, n_1) = z_M(12, n_2)$. Moreover, scanning the number of **1**s will take more steps for $x_2 = 12$ than for $x_1 = 8$, so $n_1 \neq n_2$. This shows that the partial output function is not a one-to-one function in general.

As previously stated, we want to compute some arbitrary $g(x)$ while the current state of the tape is given by $z_M(x, n)$, with x and n assumed unknown to the new Turing machine M_g. Therefore, we need a computable function h that for every x computes g based on the partial output of M_f, formally:

$$h(z_{M_f}(x, n)) = g(x). \tag{3}$$

Let us now prove that h is not computable in general (*i.e.* for arbitrary M_f and g), using the DTM shown earlier.

Theorem 1. *Let M be the DTM shown above and let $g(x)$ be any computable function such that $g(8) \neq g(12)$. Then function $h(z_M(x,n)) = g(x)$ is not computable.*

Dowód. Suppose that $h(z_M(x,n)) = g(x)$ is computable for the chosen g and M. We have already shown that, for this particular DTM, $z_M(8,n_1) = z_M(12,n_2)$ for some $n_1, n_2 \neq n_1$. Subsequently, from $h(z_M(x,n)) = g(x)$, it follows that:

$$h(z_M(8,n_1)) = g(8),$$
$$h(z_M(12,n_2)) = g(12).$$

However, we have $z_M(8,n_1) = z_M(12,n_2)$; therefore:

$$h(z_M(12,n_2)) = g(8),$$
$$h(z_M(12,n_2)) = g(12).$$

In consequence, $g(8) = g(12)$, but we assumed $g(8) \neq g(12)$. Contradiction: either $h(z_M(x,n)) \neq g(x)$ or h has two values for $z_M(12,n_2)$, meaning that h is not a function (and therefore, not computable). □

Theorem 1 states that an arbitrary change is not possible in general terms. Unfortunately, $f(x)$ and $g(x)$ are decided by the client, thus their arbitrarity is essential. We can, however, make some additional assumptions about M_f and M_g, as long as both DTMs compute $f(x)$ and $g(x)$ respectively. In particular, we can assume that M_g will be equipped with additional information concerning M_f. These may include: a) the original input x; b) the number of steps n carried before replacement; c) the internal state of M_f prior to replacement; and d) the position of the head of the machine (the currently-scanned space on the tape) prior to replacement.

The least expensive option is to let n, last internal state q_c, or last head position h_c be known to the M_g. However, we will prove that this is not enough with arbitrary f and g. Let us start with the case when n is made available to the M_g. If this still makes the partial output ambiguous, then some inputs x_1 and x_2 can lead to the same partial output in the same number of steps n. Formally, $z_M(x_1,n) = z_M(x_2,n)$. Let us construct a DTM that has this property.

Let M compute $f(x) = x + 10$ and work as follows ($\Gamma = \Sigma = \{\mathbf{1}, \mathbf{0}\}$): given x **1**s on the tape M moves along the tape until **0** is found. Since we begin at the first space of the tape and the first **0** is on $x+1$ space, reaching it takes exactly $x + 1 - 1 = x$ shifts (steps). Then M prints ten **1**s in 10 steps.

Let $x_1 = 8$. After 10 steps, M finds the first **0** in 8 steps and manages to print 2 **1**s, so $z_M(8,10) = 10$. Now, let $x_2 = 6$. After 10 steps, M finds the **0** in 6 steps and prints 4 **1**s. Thus, $z_M(6,10) = 10$. In consequence:

$$z_M(8,10) = z_M(6,10).$$

We will now use this property to show that h is not computable in general, even when the number of steps at the moment of change is known.

Theorem 2. *Let M be the DTM shown above and let $z_{M,n}(x) = z_M(x, n)$. Let $g(x)$ be any computable function such that $g(8) \neq g(6)$. Then function $h(z_{M,n}(x)) = g(x)$ is not computable.*

Dowód. For the chosen M we already showed that $z_M(8, 10) = z_M(6, 10)$. Therefore, $z_{M,10}(8) = z_{M,10}(6)$. In result $h(z_{M,10}(8)) = h(z_{M,10}(6))$, so $g(8) = g(6)$. But, we assumed $g(8) \neq g(6)$. Contradiction. □

Lastly, let us assume that M_g also knows q_c and h_c as well as the number of steps n. For this end, we define partial state and partial head position functions of a Turing machine, just as we defined the partial output:

Definition 2. *Let M be a deterministic Turing machine.* Partial state *of M is a function $q_M(x, n) : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N}$ that yields the internal state of M after n steps on input x.*

Definition 3. *Let M be a deterministic Turing machine.* Partial head position *of M is a function $s_M(x, n) : \mathbb{N} \times \mathbb{N} \nrightarrow \mathbb{N}$ that yields the number of spaces the head of M is away from its starting position after n steps on input x.*

We will now show that some DTMs will stop after a fixed number of steps, with exactly the same internal state and head position for two different inputs. Formally, we need to find a deterministic Turing machine, inputs $x_1, x_2 \neq x_1$ and number of steps n such that:

$$\begin{aligned} z_M(x_1, n) &= z_M(x_2, n), \\ q_M(x_1, n) &= q_M(x_2, n), \\ s_M(x_1, n) &= s_M(x_2, n). \end{aligned}$$

Let us note that we are not merely trying to prove that z_M, q_M and s_M are not one-to-one functions. That could mean that z_M is ambiguous only for one pair of inputs while q_M is ambiguous for some other pair of inputs. What we want to show is that all three functions can be ambiguous for the same pair of input $x_1, x_2 \neq x_1$ for some deterministic Turing machine M. We assume that the new DTM knows the number of steps, so $n_1 = n_2 = n$. We will now construct such a machine.

Let $x = \langle i, j \rangle$ be two unary numbers separated by single **0** *e.g.*:

$$x = \mathbf{11111011} = \langle 5, 2 \rangle.$$

Let M compute $f(\langle a, b \rangle) = a + b$ and let M work as follows: when M reaches the **0** separating both numbers, it will replace it with **1**. Thus, two numbers are now merged into one. However, the inserted **1** makes it that the resulting number is greater than the intended sum. To counter this, M replaces the last **1** in the entire number with **0**. Let us assume M has four states $Q = \{S, A, B, HALT\}$, $q_0 = S$. When the separating symbol **0** is reached, M replaces it with **1** and goes into state A. Then, when the

first **0** beyond the merged number is reached, M moves back one space (to the last **1**) and goes into state B. Then, M replaces the **1** with **0** and goes into the final state - halting.

Let $x = \langle 5, 2\rangle$. Thus, M will reach the separating **0** in 5 steps and then replace it with **1** on step 6 and go into state A and the head will move to the right (6 spaces from the start). Therefore:

$$\begin{aligned} z_M(\langle 5, 2\rangle, 6) &= \mathbf{11111111} = 7, \\ q_M(\langle 5, 2\rangle, 6) &= A, \\ s_M(\langle 5, 2\rangle, 6) &= 6. \end{aligned}$$

Now let $x = \langle 4, 3\rangle$. The **0** will be reached in 4 steps, and after step 5, M will replace it with **1**, go into state A, and move into the next space. In step 6, M will move one space to the right (6 spaces from the start) and will remain in state A (since the end of the second number was not yet reached). Therefore:

$$\begin{aligned} z_M(\langle 4, 3\rangle, 6) &= \mathbf{11111111} = 7, \\ q_M(\langle 4, 3\rangle, 6) &= A, \\ s_M(\langle 4, 3\rangle, 6) &= 6. \end{aligned}$$

In result:

$$\begin{aligned} z_M(\langle 5, 2\rangle, 6) &= z_M(\langle 4, 3\rangle, 6), \\ q_M(\langle 5, 2\rangle, 6) &= q_M(\langle 4, 3\rangle, 6), \\ s_M(\langle 5, 2\rangle, 6) &= s_M(\langle 4, 3\rangle, 6), \end{aligned}$$

and $\langle 5, 2\rangle \neq \langle 4, 3\rangle$ since $\mathbf{11111011} \neq \mathbf{11110111}$.

With the above DTM, we can now prove that the arbitrary changes are impossible even if the Turing machine is supplied with the values of z_M, q_M, s_M and n. This is done in a manner similar to Theorem 2.

Theorem 3. *Let M be the DTM shown above and $g(x)$ be any computable function such that $g(\langle 5, 2\rangle) \neq g(\langle 4, 3\rangle)$. Then function $h(z_M(x, n), q_M(x, n), s_M(x, n)) = g(x)$ is not computable.*

Dowód. We already showed that for this particular DTM M:

$$\begin{aligned} z_M(\langle 5, 2\rangle, 6) &= z_M(\langle 4, 3\rangle, 6), \\ q_M(\langle 5, 2\rangle, 6) &= q_M(\langle 4, 3\rangle, 6), \\ s_M(\langle 5, 2\rangle, 6) &= s_M(\langle 4, 3\rangle, 6). \end{aligned}$$

Therefore:

$$\begin{aligned} h(z_M(\langle 5, 2\rangle, 6), q_M(\langle 5, 2\rangle, 6), s_M(\langle 5, 2\rangle, 6)) = \\ = h(z_M(\langle 4, 3\rangle, 6), q_M(\langle 4, 3\rangle, 6), s_M(\langle 4, 3\rangle, 6)) \end{aligned}$$

From this and $h(z_M(x,n), q_M(x,n), s_M(x,n)) = g(x)$ it follows that $g(\langle 5,2\rangle) = g(\langle 4,3\rangle)$. But we assumed $g(\langle 5,2\rangle) \neq g(\langle 4,3\rangle)$. Contradiction. □

This shows that h is still not computable, even though it is now supplied with three arguments and n is fixed. At this point, there are still two more ways, to ensure that runtime changes are possible for arbitrary $f(x)$ and $g(x)$: 1) keep the original x on the tape; 2) impose restrictions on the structure of M_f.

The first option discards the partial output z_M. As long as the original x is still present on the tape, we can merely clear the rest of the tape and then simply compute $g(x)$, since x is given. Unfortunately, this approach still has some disadvantages:

1) we will most likely need to make a copy of x (this can be done in time $O(|x|)$, but will use at least $O(|x|)$ more space);
2) we need a method to clear the tape;
3) since $g(x)$ starts the computation from scratch, it may actually need more time than $f(x)$.

The last remark is not an issue when $f(x) \neq g(x)$ (*i.e.* slowly computed value $g(x)$ is still better than erroneous $f(x)$).

The second option assumes that there are many different M_f that compute $f(x)$, and we will restrict ourselves to those M_f that are favorable in the terms of runtime changes. It is also important for such Turing machines to be obtained in a semi-automatic manner. For example, the original DTM (program) is described by the developers and then it is transformed (by the compiler) into an equivalent version that is suitable for runtime changes.

3. Algorithm decomposition and change requirements

The first option (keeping the original input on the tape) is relatively straightforward, so let us consider the second one. Assume M is a DTM such that M computes $f(x)$. By definition, M starts computation in state q_0 and the tape has the value of x. Up to this point, we have also silently assumed that at the beginning the head of the machine is at the first space of x, denoted s_1. In other words, the starting conditions of M are as follows:

$$\begin{aligned} x &= z_M(x,0) \\ q_0 &= q_M(x,0) \\ s_1 &= s_M(x,0) \end{aligned}$$

Let us assume that M has carried out n steps. The remainder of the steps of M are equivalent to some other DTM, denoted by M', that has the same definition as M except the initial state and the starting conditions:

$$\begin{aligned} x' &= z_M(x,n) \\ q'_0 &= q_M(x,n) \\ s'_1 &= s_M(x,n) \end{aligned}$$

Therefore, for a given x, M can be decomposed into a set of N different DTMs $dM_x = \{M_1, M_2, \dots M_N\}$ with starting conditions of M_k depending on the final conditions of M_{k-1} for $k > 0$. Let us also note that dM_x can be different for various x; moreover, a general dM (for each x) may not be known, since determining the required numbers of steps for a DTM to halt requires determining whether a given DTM halts or not (the halting problem), which has been proven to be undecidable. Therefore, we will propose a decomposition method for Turing machines that is beneficial for runtime changes in this chapter, and we will briefly describe possible types of changes and their requirements.

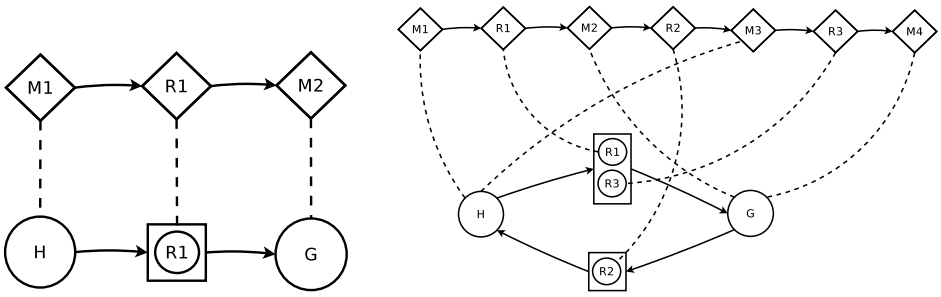
Let us start by stating that every deterministic Turing machine M can be thought of as a set of Turing machines (subDTMs), *i.e.* $M = \{M_A, M_B, \dots\}$. SubDTM M_A contains a set Q_A of states from the original machine, *i.e.* $Q_A \subset Q$. Moreover, $Q_A \cup Q_B \cup \dots = Q$. In the most basic case (when no decomposition is available), we have $M_A = M$ and $Q_A = Q$.

Our decomposition method is based on dividing the original Turing machine (computable function) into a set of subDTMs (called subtasks) in such a way that the state of the tape between every two subDTMs is in some kind of *expected format*. For example, let us consider a quick sort procedure which roughly work as follows: a) choose a guardian element; b) move elements lower than the guardian before it; c) move elements greater than the guardian after it; d) sort elements before the guardian; and e) sort elements after the guardian. These steps clearly describe 5 subDTMs (however, further decomposition might be possible), since we can reason about the state of the sorted array after each step. For example, after step d), the elements up to the guardian element are always sorted. If we tried to divide d) into two subDTMs, however, we wouldn't be able to predict what the sorted array looks like in the middle of step d), unless some further assumption were made.

If the above method is applied, we obtain a graph of subtasks (which correspond to subDTMs or computable functions and relations between them) capable of performing the original task, as the expected formats between subtasks are known. The expected format before and after a subtask can be called a *protocol* of this subtask, and is similar to the concept of provided and required interfaces known from the objective programming paradigm. The only thing that is missing is a set of subDTMs that will reposition the head of the machine so the next subtask will start processing exactly where its needs to. These subDTMs will be called *repositions*. A reposition is not allowed to modify the contents of the tape (it has to write on the tape the same symbol it read from it). In this case, the subtasks and repositions roughly correspond to the functions and function calls in programming languages like `C/C++`. Some examples of such decompositions are shown in Figure 1. From this figure, we can also conclude that a single subtask can be "called" in more than one context (using different repositions) and than one subtask can be "implemented" by many subDTMs.

The above decomposition is similar to the graphs of components mentioned in the introduction, but more general. A single subtask can correspond to a function

(or method), software component, the entire program (or a single thread). Loops, conditional statements, and code blocks can be defined as subtasks. In some cases, even single statements can work as a single subtask. This is a much more powerful concept than graphs of components, as it offers a higher degree of control and is independent from the programming languages or paradigms used. Moreover, one subtask can be composed of several smaller components (called its children), which further enhances the flexibility of this method. An important thing to note is that the choice of decomposition is left to the programmer or to some automatic tools, therefore allowing different decompositions for the same DTM and remaining flexible.



(a) $f(x) = g(h(x))$ (b) $f(x) = g(h(g(h(x))))$

Figure 1. Examples of decompositions for Turing machines. H and G are subtasks. R_1 to R_3 are repositions. M_1 to M_4 are subDTMs.

As stated before, every deterministic Turing machine can be decomposed with this method. In the worst-case scenario, we will obtain a single subtask with no repositions. With a graph of subtasks present, we can now define the runtime change in terms of changes done to the graph of subtasks. A subtask to be changed can be simply replaced with other subtasks, as long as some requirements are met. We will now briefly consider these requirements for several possible types of runtime change.

A basic type of change is an inner change of subtask, *i.e.* subtask realizes the same computable functions as before but in a different way (with different algorithm). That means that the protocol of the subtask remains unchanged and does not affect any other subtask. However, the subtask can change its starting or ending tape position, possibly forcing changes in all repositions connected to this subtask. In result, the changes cannot take effect as long as the affected subtask or any of the affected repositions are executing. Note that, parent-children subtasks aside, only one subtask or reposition can execute at a time.

Another type of change is when a subtask changes the computable function it realizes, therefore changing either its expected input format, its expected output format, or both. This forces changes in all subtasks (and their repositions) which are connected to the given subtasks, so the input and output formats of the connected subtasks match. This defines a range or set of subtasks that cannot execute for the change to occur. Examples of such ranges are shown in Figure 2. We can also consider

subtask removal or addition, but those cases are similar to the change of the input protocol of a subtask and impose the same requirements. Other types of runtime changes may be possible as well.

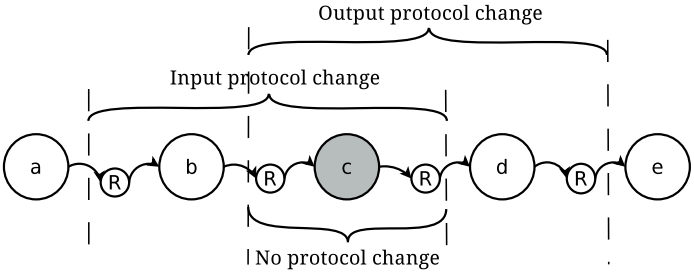


Figure 2. Range of subtasks that cannot execute for the subtask c to change safely.

Finally, let us note that the above-mentioned requirements only define which parts of the system cannot execute for the change to occur safely. In particular, we assume that the proposed changes are logical, *e.g* each pair of connected subtasks has its output and input protocols matched. The task of supplying a valid change schedule is left to the developer and automation tools (*i.e.* compilers).

4. Runtime adaptation model

In this section, we will propose a new model of computation with the ability to apply runtime changes by extending the existing model of Universal Turing Machine. We will start by describing our version of the UTM, and then we will extend it to create a dynamic version and facilitate runtime changes.

Our basic UTM has three bidirectional, infinite tapes. All tapes have independent tape heads, and only one tape head can shift per step (a less-restricted model is possible where at least one head tape shifts every step). The first tape is used for processing, *i.e.* it contains input in the beginning and output in the end. We limit ourselves to a binary alphabet of **1**s and **0**s. This limitation does not reduce the number of problems solvable by our UTM, as every problem can be encoded using binary representation – the size of the alphabet can be reduced as long as the number of states of the machine is increased in response.

Since UTMs work by simulating other DTMs, we need a description of a DTM to simulate. This is precisely what the second tape is used for. We use an alphabet of 9 symbols: D, B, A, R, L, 0, 1, H, and blank symbol b to store information about the simulated Turing machine, *i.e.* its states and transition function. Exemplary description of a state may look as shown in Figure 3.

Symbol D starts the state description and is followed by a number of As, which define the state number. In the above example, we have four As (AAAA), therefore this is state number 4. Similarly, DA is state 1 and D is state 0. Next, we have

Figure 3. Exemplary state description for Universal Turing Machine.

two sections describing actions to be taken when **0** or **1** are scanned on the current space of the processing tape. In our example, we have R0BAA when 0 is scanned and L1BAAAAA when 1 is scanned. 0 and 1 indicate what symbol to write in the current space (on the processing tape), R and L indicate whether to shift to the right or left (on the processing tape) and the last element (BAA and BAAAAA) indicate what state to go to next (in this case, BAA means state 2 and BAAAAA means state 5). This, however, describes a non-final state. An example of a final state looks like this: DAAH. This would be state 2, which is at the same time a final state. The entire DTM program is simply a string of all its states. For example, a 3-state DTM that searches the input for the second **1** and then halts would have a program as follows:

DR0BR1BADAR0BAR1BAADAAH.

The UTM simulates the target Turing machine as follows: the head on the program tape starts at the beginning of the first state description and shifts to position itself after the state number (after the "DAA" sequence). Then, the symbol on the processing tape is scanned and the program head may shift again to reach the first symbol of the corresponding section (for **0** or **1**). Then, the information in the correct section is used to write the desired symbol on the processing tape and shift the processing head left or right. Now, all that is left is to reach the next state.

For this purpose, the third tape (the target state tape) is used. After the symbol on the processing tape was written and the tape was shifted, the number of the next state (for example "AAA") is copied into the target state tape. After that, the program head is shifted to the right until the next state candidate is found (indicated by reaching symbol D). Then, the target state tape is used to check whether the candidate state is the desired one. The check is performed simply by moving the head of the target state tape to the beginning and then checking step-by-step whether the number of As in the candidate state and on the target state tape matches. If not, we shift right once more in search for another candidate state. When the end of the program is reached, then we perform the search in the opposite direction until the desired target state is found. Finally, when the state is found, the program head is positioned once again after the state number (the "DAA" sequence) and the cycle repeats itself. If a halting state is entered (indicated by H symbol after the "DAA" sequence), then the machine halts.

The described UTM can simulate any DTM with a binary alphabet and works much slower then the DTM it simulates, as in each cycle a part (possibly considerable) of the program must be traversed. The original UTM researched by Turing meant the smallest possible UTM (*i.e.* the smallest alphabet size or number of states), but our

aim was not to discover the best UTM out there – at the moment, we prefer less efficient yet more clear examples.

Now we will define our runtime adaptation model of computation by extending the above-mentioned UTM to create a Dynamic Universal Turing Machine (DUTM). Let us start with the source of the changes. To this end, we add another (fourth) tape which will be called the change tape. When no changes are requested, this tape remains empty. When the requests appear, then two pieces of information will appear on this tape: 1) the list of subtasks that cannot execute for the change to occur; and 2) the new program for our DUTM. The first part follows the previous convention: if the subtasks 2 and 5 cannot execute at the moment of change, then this part will take the following form: CAACAAAAA. The symbol C is therefore added to the alphabet to indicate subtasks (where D and B indicate states and target states). If another request is made, then it is added after the first one.

In order to apply changes, our DUTM needs to check for new content on the change tape. It is logical to do so only when the currently-executed subtask changes. Therefore, we modify the format of the program. States that are not starting states for subtasks remain as before. States that start a new subtask will receive another part which describes which subtask has just begun. If we used the exemplary state from before and define that this state starts subtask 3, then the state description will look as follows:

DAAAACAAAR0BAAL1BAAAAA.

With this information, we can modify how our DUTM works. In each cycle, we were positioning the program head on the **0** or **1** section. If during this process symbol C is scanned, then this indicates that a new subtask was reached. Therefore, we use another (fifth) tape to copy the current subtask number ("AAA" for subtask 3). Let us call this tape the subtask tape. After this, we check the change tape for changes. If no changes are present, we carry on as before (write symbol, shift processing tape, go into the next state). If changes are present, we check whether the current subtask is in the list of "forbidden subtasks". If we find that our subtask is in the list (the number of As in our subtask and in the current one from the list is equal), then the changes cannot be applied now and we carry on as before. If we reach symbol D instead (meaning we went past the list of "forbidden" subtasks), then the change can be applied safely.

The process of applying changes is as follows: the number of the current state is stored on the target state tape. Then, the program tape is cleared (*i.e.* we shift to the end of the tape and erase all symbols until we reach the beginning). After this, the new program from the change tape is copied into the program tape. Finally, we need to move to the state we stored on the target state tape. For this purpose, we use the next state-searching procedure from before. When this step is completed, we reach a new cycle (program head above the **0** section), and the program tape has the desired new contents.

Formally, we defined a 5-tape deterministic Turing machine with an alphabet of 10 symbols $\Gamma = \{A, B, C, D, R, L, 0, 1, H, b\}$. The set of states and the transition

function is too complex to present here, so we will limit ourselves to the flowchart of the DUTM, which is shown in Figure 4.

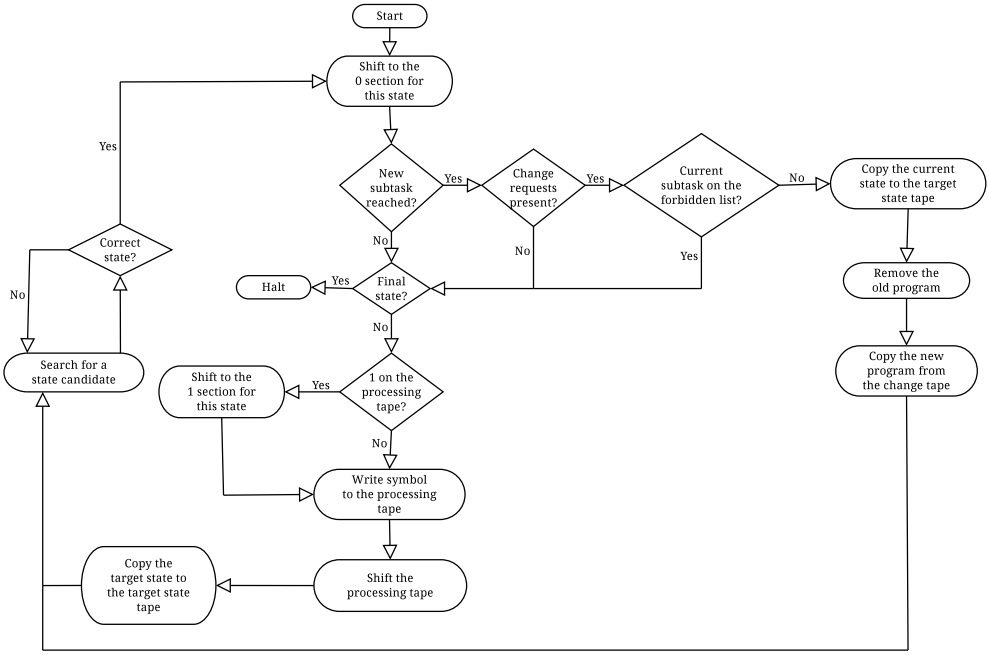


Figure 4. Flowchart for the Dynamic Universal Turing Machine.

Let us note that the above-mentioned model assumes that all changes are correct, *i.e.* the supplied new program is valid and has passed all necessary checks from the developers and the development tools (compilers). Our model is only aimed at applying the specified runtime changes safely and allowing possibly arbitrary changes. The list of the "forbidden" subtasks should be valid as well, and its creation is left once more for the developers or the automation tools.

5. Conclusions and future work

In this paper, the authors presented research on the possibility of applying the concept of models of computation to the problem of runtime software changes. The possible types of runtime changes were represented by changes to computable functions and Turing machines. This allowed for applying this to every software system that conforms to the Turing machine model. In result, this concept could be used with a wide range of programming languages and paradigms that are based on Turing machines and equivalent models of computations.

This paper also presents a method of decomposition for Turing machines into graphs of subtasks and defining runtime changes through certain transformations of

those graph. Different transformations were considered, and certain prerequisites of runtime changes (namely, subtasks execution conditions during given change) were studied and described as well.

Finally, a new model of computation, the Dynamic Universal Turing Machine, was proposed as an extension to the Universal Turing Machine model known from the literature. This model is capable of recognizing the changes requested by an external source (*i.e.* developers) and apply them as soon as possible without the need to shutdown or restart the machine. This model is an extension of the classic UTM model, therefore it is capable of solving every problem that is solvable using the original UTM model (even though the alphabet of the simulated Turing machine was limited to two symbols for convenience).

The study of the theoretical and practical properties of the proposed model, especially in comparison with the basic Turing machine, still remains an open problem. The same applies for the possibility of performing runtime changes in similar models of computations like RAM or RASP.

References

[1] Bennett C. H.: Logical Reversibility of Computation. *IBM J. Res. Dev.*, **vol. 17**(6), pp. 525–532, 1973. ISSN 0018-8646. http://dx.doi.org/10.1147/rd.176.0525.

[2] Church A.: *An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics*, **vol. 58**(2), pp. 345–363, 1936. ISSN 00029327. http://dx.doi.org/10.2307/2371045.

[3] Hopcroft J., J.D. Ullman: *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Cambridge, 1979.

[4] Kleene S. C.: *Introduction to Metamathematics.* North Holland, 1952.

[5] Mukhija A., Glinz M.: Runtime Adaptation of Applications Through Dynamic Recomposition of Components. In: *Proceedings of the 18th International Conference on Architecture of Computing Systems Conference on Systems Aspects in Organic and Pervasive Computing*, ARCS'05, pp. 124–138. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 3-540-25273-8, 978-3-540-25273-3. http://dx.doi.org/10.1007/978-3-540-31967-2_9.

[6] Rosser B.: An Informal Exposition of Proofs of Gödel's Theorems and Church's Theorem. *The Journal of Symbolic Logic*, **vol. 4**(2), 1939. ISSN 00224812. http://dx.doi.org/10.2307/2269059.

[7] Rudy J.: Runtime software adaptation: approaches and a programming tool. *Journal of Theoretical and Applied Computer Science*, **vol. 6**(1), pp. 75–89, 2012. ISSN 2299-2634.

[8] Turing A.: On Computable Numbers with an Application to the Entscheidungs Problem. In: *Proc. London Mathematical Society*, **vol. 2**(42), pp. 230–265, 1936.

[9] Valetto G., Kaiser G.: A Case Study in Software Adaptation. In: *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pp. 73–78. ACM, New York, NY, USA, 2002. ISBN 1-58113-609-9. http://dx.doi.org/10.1145/582128.582142.

[10] Valetto G., Kaiser G. E., Kc G. S.: A Mobile Agent Approach to Process-Based Dynamic Adaptation of Complex Software Systems. In: *Proceedings of the 8th European Workshop on Software Process Technology*, EWSPT '01, pp. 102–116. Springer-Verlag, London, UK, UK, 2001. ISBN 3-540-42264-1.

[11] Villazón A., Binder W., Ansaloni D., Moret P.: Advanced Runtime Adaptation for Java. *SIGPLAN Not.*, **vol. 45**(2), pp. 85–94, 2009. ISSN 0362-1340. http://dx.doi.org/10.1145/1837852.1621621.

[12] Wang Q., Huang G., Shen J., Mei H., Yang F.: Runtime Software Architecture Based Software Online Evolution. In: *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, COMPSAC '03, pp. 230–. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-2020-0.

Affiliations

Jarosław Rudy
Institute of Computer Engineering, Control and Robotics, Wrocław University of Technology, Janiszewskiego 11–17, 50-372 Wrocław, Poland, jaroslaw.rudy@pwr.wroc.pl

Received: 31.01.2014
Revised: 31.03.2014
Accepted: 01.04.2014