Computer Science • 17 (2) 2016 http://dx.doi.org/10.7494/csci.2016.17.2.187

Jarosław Rudy

DYNAMIC TURING MACHINE: MODEL AND PROPERTIES FOR RUNTIME CODE CHANGES

Abstract *In this paper, a dynamic model of computation based on the Universal Turing Machine is proposed. This model is capable of applying runtime code modifications for 3-symbol deterministic Turing Machines at runtime and requires a decomposition of the simulated machine into parts called subtasks. The algorithm for performing runtime changes is considered, and the ability to apply runtime changes is studied through computer simulations. Theoretical properties of the proposed model, including computational power as well as time and space complexity, are studied and proven. Connections between the proposed model and Oracle Machines are discussed. Moreover, a possible method of implementation in real-life systems is proposed.*

Keywords computability theory, models of computation, turing machine, runtime code changes

Citation Computer Science 17 (2) 2016: 187–224

1. Introduction

During the first half of the 19th century, extensive research concerning the nature of computation and computability was carried out. This resulted in the development and study of various models of computation, including the Turing Machine [16]. This, coupled with the invention of the modern computer years later, led to the constant development of software, and this process has continuously increased in intensity (especially over the past decade). Software in its various forms has affected? almost every aspect of human life and is accessible to a wide range of customers.

However, with the widespread use of software, the problems facing its development have changed. In the modern, fast-paced world, the requirements of the clients or end users often change; this issue cannot be completely resolved with the use of software engineering and requirement analysis due to the human factor. Moreover, software often needs to be modified because of the changing legal regulations or in response to programming errors and security issues.

This issue is commonly resolved by using software updates or plug-ins, but those usually apply changes by restarting the application. It is rare and more interesting to consider changes done at runtime without the need to stop the application, potentially preventing a loss of data. The possibility of such a runtime change has been discussed in the literature to some extent. Most approaches use the concept of models of software in order to represent the system and define the runtime changes in more abstract and manageable terms. Thus, the target system and model are tied together, so the changes in the former can be reflected in the latter, and vice versa. Examples of such apporach include papers by Wang et. al [19], Cheng et. al [3], and Garlan et. al [8]. Another approach is the use of software architectural styles – in particular, styles that make a runtime change easier. Such an approach was proposed by Oreizy et. al [11]. The disadvantage is that the target system must conform to the given architectural style; thus, greatly limiting its form and subsequently causing problems for already-existing legacy systems that were designed in a different way. Moreover, various approaches for specific programming languages and paradigms exist, including Aspect-Oriented Programming [12], software agents [17], and Java [18]. Among the approaches for Java is a paper by Dmitriev [6] in which the runtime changes are applied with the help of a debugger as well as similar low-level tools for Java. This paper also defines four levels of runtime changes, with level 1 being changes to the method body and level 4 being arbitrary changes. This approach is successful in implementing only the first level of runtime changes, proving that there is still much to research in the field of which changes are possible and how they are to be applied. Another solution for Java was proposed by Zhang et. al [21]. These authors treat a change as a path on a graph of safe configurations, allowing us to better judge whether a given change is possible or not. As a last possibility for Java, we will mention the OSGi system, which can be used to enrich the Java environment into a dynamic platform of components (with a heavy emphasis on modularity and services).

And last, we would like to describe two approaches that are not destined for Java. The first solution, described in [13], is intended for C applications running under the Linux operating system (although extension to other systems is possible). The solution works by using the dynamic loading of libraries and moving those parts of the code that are meant to be dynamic into separate libraries. Interestingly, the system keeps the previous code versions, and in the event of a runtime error, it is possible to use the older code version instead of the new one (which allows for immediate function resumption. The second approach is the POLUS system [2], which can be used to dynamically evolve running applications into their never versions. This system must be supplied with the object code for both versions before it can produce a patch file, and then a dedicated application running in the background can apply the change. The approach itself is fairly powerful, and special care was taken to avoid incoherence; *i.e.,* POLUS guarantees that the new function versions won't call the old ones. The research also indicates that POLUS doesn't incur a significant performance drop, although the research presented is limited. Two drawbacks of POLUS are the fact that the changes are limited to functions and that POLUS cannot be used to make changes for interpreted languages like Java or Perl (since it *can* evolve the Java Virtual Machine or the Perl interpreter, but it *cannot* evolve programs currently running interpreted).

The above literature review can be used to conclude that most of the existing approaches focus on specific applications driven by practice, with little emphasis on theory and properties of runtime changes. In particular, the existing approaches are incompatible with each other and are not general, as most of them are dedicated to be used for only one programming language (usually C/C++ or Java). The possibility of extending the ideas presented in this literature to other similar languages is rarely discussed, thus making it questionable. Moreover, the approaches usually support limited programming paradigms (most common is object-oriented programming) or may even *require* other paradigms (like software agents or aspect-oriented programming) to function properly, which clearly limits the applications of such solutions.

The next important point is the study of the very nature of dynamic changes. Several programming languages are supplied with features that make dynamic changes easier. These features include the presence of a virtual machine or reflection mechanism (Java and Smalltalk are examples of languages that rely heavily on such features). In general, there exists a notion of dynamic programming language. Such languages allow many actions that are normally carried out during compilation to be performed at runtime. Thus, in theory, it seems that languages (environments) like Java and C# are supplied with everything they need to make runtime changes possible. However, dynamic languages such as these are not explicitly designed to support runtime changes. As a result, what we lack is the know-how – the practical knowledge on how to perform runtime changes using such mechanisms as reflection. Furthermore, we would like to perform changes even in languages that do are not dynamic and, thus, do not possess such features as reflection. To sum it up, we are clearly lacking a coherent theory for representing, verifying, and applying runtime changes in a general way, independent from the used programming language or paradigm.

Despite this, some theoretical research does exist, including a paper by Zhang et. al [20] with studies of the so-called quiescent states and global invariants of the target system. Another concept of representing runtime changes with the use of models of computation was proposed by Rudy [14]. Runtime changes can be thought of as changes made to the computable function being computed or the algorithm computing it, therefore affecting both the output and the computational complexity of the system. A method of decomposition for *deterministic Turing Machines* (DTM) was introduced with the intention of dividing the original machine into a set of so-called *subtasks*, each representing a different computable function or algorithm. The rule of the decomposition is that when each subtask completes, it leaves the tape in some kind of *expected format* (protocol), so the next subtask can continue the computation. A set of special subtasks called *repositions* is used to shift the head of the machine on the tape between subtasks. Repositions cannot modify the tape. With such a decomposition method defined, an algorithm for performing runtime changes was outlined with an emphasis on the change requirements; *i.e.,* which subtasks cannot execute for a give subtask to change safely.

This paper, we aim to extend and refine the model of computation outlined above and study its theoretical and practical properties. In particular, we would like to define a dynamic model of computation that could be used to design software with runtime-code-change capability for a wide range of programming languages and paradigms. We hope that the researched properties (especially time and space complexity) will allow us to better understand the possibilities and limitations of the runtime code changes and (in the future) help us design more practical systems for a wide range of programming languages and paradigms.

The remainder of this paper is organized as follows: Section 2 describes the special versions of the Deterministic Turing Machine and Universal Turing Machine used in this paper, along with their basic properties. In Section 3, a new dynamic model of computation based on the special case of the Universal Turing Machine is proposed. Section 4 contains the study of the theoretical properties of the proposed model, including time and space complexity with relation to the standard Universal Turing Machine. Section 5 offers a discussion. Finally, Section 6 contains conclusions as well as a brief comparison of different models of computation.

2. Turing Machines

This section serves as a necessary foundation for the definition of the dynamic model of computation, and is composed of two major parts. The first part defines the base Turing Machine model used through the rest of the paper, including all of the differences from the commonly used Turing Machine. The second part includes the definition of our version of the Universal Turing Machine, which can be used to simulate the machine from the first part and will serve as the basis for the dynamic model.

2.1. Deterministic Turing Machine

The original Deterministic Turing Machine (DTM) was developed by Alan Turing [16]. Various modifications of the DTM have appeared in the literature; thus, we have decided to use the version by Hopcroft and Ullman [9] as a basic definition of the DTM. This DTM assumed two finite, non-empty sets of symbols Γ and $\Sigma \in \Gamma$, called the tape alphabet and input alphabet, respectively, and one symbol b was called the blank symbol. Moreover, the DTM had two finite, non-empty sets of states Q and $F \in Q$, called the internal states and the final states, respectively. One state q_0 was called the initial state. The last element was the transition function δ. Thus, any DTM M can be formally defined as a 7-tuple $M =< Q, \Gamma, b, \Sigma, \delta, q_0, F >$.

The DTM had one bidirectional tape with unlimited number of cells (spaces), each in the beginning occupied by a single symbol either b or $s \in \Sigma$, and the blank was the only symbol allowed to occupy an infinite number of spaces. The DTM also had one scanning head, which was positioned at any given time over one symbol (called the current symbol) of the tape and could move left or right by one space.

The computation was carried out step-by-step with the use of the transition function δ. For each pair of current tape symbol Σ and current non-final state $Q \setminus F$, the function defined three elements: 1) the symbol to write on the current space on the tape; 2) the direction for the head to shift (either left or right); and 3) the internal state to go to. This process continued until any final state from F was reached, at which point the DTM halts. The state on the tape (once halted) is the answer produced by the DTM. This model was commonly associated with two complexity measures: $T(n)$ indicated the number of steps needed for the DTM to halt on input n. Similarly, $S(n)$ indicated the number of spaces on the tape needed for the DTM to produce an answer for input n.

In order to make the discussion more clear and understandable, we will modify the previous description of the DTM by making a few alterations. However, we will later prove that the resulting modification of the DTM has exactly the same computational power as the DTM presented above. First, we combine both arbitrary symbol alphabets Γ and Σ into one fixed 3-symbol alphabet $\Gamma = \{\mathbf{0}, \mathbf{1}, \#\}$, with $\#$ being the blank symbol. Second, we assume that the set F of final states has only one element f. The third change is the addition of two new tapes called the *input tape* (T_{IN}) and *output tape* (T_{OUT}). The original tape is called the *data tape* (T_{DATA}). T_{IN} contains the input for the DTM, and the rest of its spaces are filled with blanks ($\#$). Two other tapes are filled with blanks in the beginning. Each tape has a tape head called H_{IN}, H_{OUT}, and H_{DATA}, respectively. The tape heads move independently.

In each step, the machine performs one of four possible operations: input, output, halt, or process. If the final state f is reached, the DTM halts as before, but the output produced is read from T_{OUT} only. The input operation occurs when the current symbol from T_{IN} is copied into the current space of T_{DATA}. After this, the internal state of the DTM may change, and H_{IN} shifts one space to the right while the other heads stay intact. Similarly, the output operation consists of copying the current symbol from T_{DATA} into the current space of T_{OUT} and then shifting H_{OUT} one space to the right and possibly entering a new internal state. The processing operation is exactly like the step from the original DTM: the current state on the T_{DATA} is overwritten, H_{DATA} shifts either left or right, and a new internal state may be entered. The structure of such a DTM is shown in Figure 1. We also assume that, in the beginning, H_{IN} is placed at the first non-blank symbol on T_{IN} (starting from the left side).

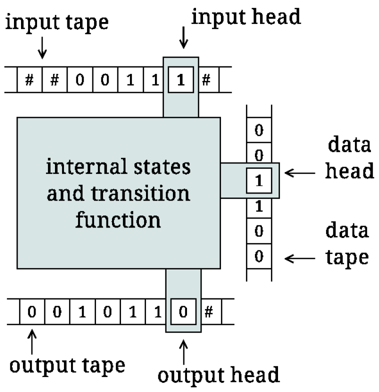

Figure 1. The structure of the 3-symbol on-line deterministic Turing Machine.

The changes to the alphabet and final states are done mostly for convenience, as this allows for a simpler and clearer definition of the Universal Turing Machine later. The changes to the tapes, however, are meant to follow the concept of an on-line Turing Machine, where a part of the input may be processed and output produced before the next part of the input is accessed. This is useful to model some properties of software (in particular, servers and interactive applications). In this case, T_{IN} can be thought to contain client requests or be a keyboard buffer. On the other side, T_{OUT} can be interpreted as server responses or an output file.

Let us note that the heads of T_{IN} and T_{OUT} can shift only to the right, meaning they act more like streams than typical DTM tapes, and each position of the T_{IN} and T_{OUT} can be read from or written to only once. Of course, the DTM is free to make as many copies of the input on T_{DATA} as needed. The advantage of such system is that, when some output is produced, the T_{DATA} does not need to be erased – the H_{DATA} may just be shifted to a new, untouched portion of the tape and then continue processing new input.

Now we formally define our 3-symbol on-line Deterministic Turing Machine M (referred to simply as DTM3 throughout the remainder of this paper) as 6-tuple $M = \langle Q, \Gamma, \#, q_0, f, \delta \rangle$ as follows:

- Q – finite non-empty set of internal states,
- $\Gamma = \{\mathbf{0}, \mathbf{1}, \#\}$ – set of tape symbols (tape alphabet),
- $\# \in \Gamma$ – the blank symbol,
- $q_0 \in Q$ – the initial state,
- $f \in Q$ – the final state,
- $\delta : Q \setminus \{f\} \times \Gamma \to Q \times \Gamma \times \{L, R, I, O\}$ – the transition function.

The transition function is defined for each pair of tape symbol and non-final state. The function determines the symbol to write to T_{DATA}, next internal state to move to, and the shifts for the heads: L (R) shifts only H_{DATA} to the left (right), while I (O) shift only H_{IN} (H_{OUT}) to the right.

The above DTM3 is defined quite differently from a typical Turing Machine found in the literature. Therefore, we would want to prove that, despite this, it remains just as powerful as the original Turing Machine; *i.e.,* it can compute the same (or greater) class of computable functions and consequently solve the same class of problems. This is done via Property 1.

Property 1. *For every computable function f, there exists DTM3 M_3 that computes f.*

Proof. For every n-ary computable function $f(a_1, a_2, \ldots, a_n) : \mathbb{N}^n \to \mathbb{N}$, there exists function $f_n(a) : \mathbb{N} \to \mathbb{N}$, such that $f_n(\pi^{(n)}(a_1, a_2, \ldots, a_n)) = f(a_1, a_2, \ldots, a_n)$, where $\pi(n)$ is the Cantor tuple function. This follows from the definition of $\pi^{(n)}$. Therefore, it is sufficient to show that there exists M_3 for every $f_n(a) : \mathbb{N} \to \mathbb{N}$, and it computes $f_n(a)$. From the Church–Turing thesis (starting with Kleene [10]), it follows that, if $f_n(a)$ is a computable function, then there exists a regular DTM M that computes it. Since every natural number can be encoded over the alphabet $\{\mathbf{0}, \mathbf{1}\}$, let the alphabet of M be $\Gamma = \Sigma = \{\mathbf{0}, \mathbf{1}, \#\}$. Let M_{IN} be a DTM3 that copies its T_{IN} into T_{DATA} and then halts. Similarly, let M_{OUT} be a DTM3 that copies its T_{DATA} into T_{OUT} and then halts. Now, let our M_3 be a composition of the above three:

$$M_3 = M_{\text{IN}} \mid M \mid M_{\text{OUT}},$$

i.e., when a machine in the composition halts, the next machine starts in its own initial state with the contents of the tapes left behind from the previous machine. Therefore, M works as a regular DTM on the input supplied by M_{IN}, and the final output is produced by M_{OUT}. Therefore, we constructed a DTM3 with its input on T_{IN} and its output on T_{OUT} that computes $f_n(a)$. □

2.2. Universal Turing Machine

The DTM3 presented above is a class of Turing Machines we would like to equip with support for runtime changes. Turing Machines on their own, however, leave no possibility for changing the program, since the set of internal states Q and the transition function δ do not change in any way during computation. Fortunately, there exists a concept of the Universal Turing Machine (UTM) introduced by Turing in his original paper [16]. The UTM is a Turing Machine that is able to simulate any other Turing Machine (including itself). This is done by encoding and storing the simulated DTM on the UTM as data. Since the encoded DTM is just a set of symbols on the tape, it can be modified and processed as any other data. Moreover, by UTM, Turing understood the simplest (by some criteria) machine capable of simulating all Turing Machines.

In this subsection, we will present our own version of UTM for simulating any DTM3 (defined above), called the DTM. This machine will serve as the basis for the dynamic computation model presented in the next section. It is important to note that the UTM3 will not be a true UTM. First, it will be able to simulate only 1-, 2-, or 3-symbol DTMs. However, this restriction is meant (as before) to simplify the design process and make the discussion clearer, and a method for extending this into UTM for any (on-line) Turing Machine will be presented. Moreover, our UTM3 will most likely not be the simplest of its kind; but once again, we focus on simplicity.

The UTM3 needs a way to store the internal states and transition function of the DTM3 it simulates on one of its tapes. T_{DATA} can be used for this, but it would be problematic to have one tape serving both purposes. Therefore, we add another tape called the *program tape* (T_{PROG}) with its own head H_{PROG}. The program stores the information about all states of the DTM3 in which to simulate. A description for each state starts with the state number section, and this consists of the symbol D followed by a number of As, so state 0 is D, state 1 is DA, state 2 is DAA, and so on. The next section of the state description depends on the state type:

- For the final state (the last state by convention) this section simply contains one H symbol. Example state description is DAAAH (state 3, halting).
- For the input states, the second section contains symbol I, followed by the designation of the target state (*i.e.,* state to go into). This is done by single symbol B followed by the number of As (B, BA, BAA etc.). Example state description is DAAIBAAA (state 2, input, go to state 3).
- The output states are just like the input states except for the symbol O instead of I. Example state description is DAAAAOBA (state 4, output, go to state 1).
- For all other states, this section is divided into three subsections: one for each DTM3 tape symbol. Each subsection consists of three parts: the symbol to write on T_{DATA} (either **0**, **1** or #), the shift direction for H_{DATA} (either L or R), and the target state designation (B, BA, BAA etc.).

An example of such state encoding for the normal state is shown in Figure 2. This is the description of state 4. Subsections mean "if **0** then write **1**, shift right and go into state 2", "if **1** then write #, shift left and go into state 5" and "if # then write **0**, shift left and go into state 3".

We assume (as in the case of DTM3) that, in each state, the UTM3 can shift as little as one and as many as all of its tape heads. The UTM3 works as follows: at the beginning, H_{PROG} is placed on the D symbol of state 0 (since it is the first state by convention). This state has zero number of A's, so after a single shift of H_{PROG}

to the right, the UTM3 is able to determine the type of this state (H for halt, I for input, O for output, others indicate normal state).

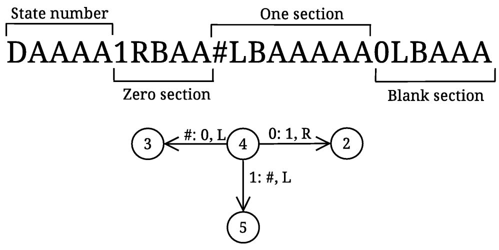

Figure 2. Example of a state description encoding for the UTM3.

For the halting state, the machine halts at this point. For input states, the UTM3 copies the symbol from the space under H_{IN} into the space under H_{DATA}. After that, H_{IN} is shifted to the right, and the procedure for reaching the next state (described below) is started. Output states are exactly the same except that the symbol beneath H_{DATA} is copied into the space beneath H_{OUT} and then H_{OUT} is shifted to the right. For normal states, the UTM3 first shifts to the correct subsection (either for zero, one, or blank) and then overwrites the symbol under H_{DATA} with the current symbol under H_{PROG} (which is either **0**, **1**, or #). Then, H_{PROG} shifts to the right, where the intended shift direction (either L or R) is stored. Next, H_{DATA} is shifted according to this direction. Finally, the procedure for reaching the next state is started.

The procedure for reaching the next state is always started with H_{PROG} placed directly left to the target (next) state designation (*e.g.,* BAAA). In order to reach the intended state, the designation of the state needs to be stored. To this end, we equip the UTM3 with one more tape called the *state tape* (T_{STAT}, H_{STAT}). This tape is initially empty (*i.e.,* filled with blanks).

With this tape defined, the state-reaching procedure works as follows: we shift H_{PROG} to the right until symbol A is found, then we copy all subsequent A symbols to the T_{STAT}. After this, T_{STAT} holds the intended state; *e.g.,* AAA if state 3 is intended. Next, we search for a state candidate by shifting H_{PROG} to the right until the next state (symbol D) is found. Then, we check whether this candidate state is the intended state. The check is done by positioning H_{PROG} at the first A symbol of the candidate state and H_{STAT} at the first A symbol of T_{STAT} and then checking whether the number of A's matches. If both strings of A's end at the same time, then the candidate state is indeed the intended state, and the procedure ends. Otherwise (substrings of A's are different), this means that the candidate state is not the intended state. In this case, we simply shift H_{PROG} once again to the right in search of another candidate state. However, if we encounter double blank symbol (##), this means that we have reached the end of T_{PROG}. This can happen when the intended state to go into is placed earlier on the tape than our position at the beginning of the procedure. In this case, we need to simply reverse the search direction.

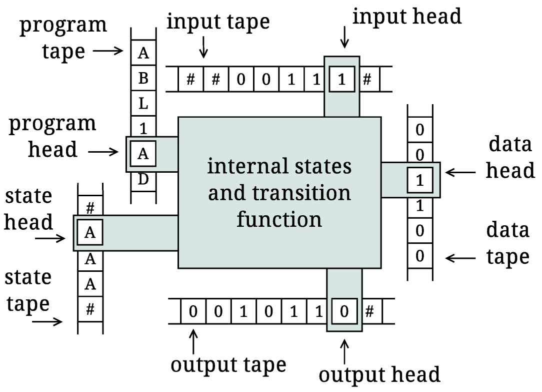

Figure 3. The structure of the 3-symbol on-line deterministic Universal Turing Machine.

Assuming that T_{PROG} contains valid DTM3 encoding, then the intended state will eventually be found. When the procedure ends, we clear the contents of T_{STAT} by filling it again with blanks. Let us recall that, if the candidate state was the correct one, then that means we went past the string of A's on T_{PROG} at the exact same time we went past it on T_{STAT}. In result, H_{PROG} is now placed directly right of the state designation, where the state type is stored. At this point, we have simulated one step of the target DTM3, and we start the next step with H_{PROG} already placed to instantly determine the type of the current state.

With this, we have constructed a Deterministic Turing Machine capable of simulating any valid DTM3. The structure of the UTM3 is shown in Figure 3. It is important to note that, unlike DTM3, the UTM3 is a single machine. Before we conclude this subsection, we will establish a few properties of the UTM3. First, the UTM3 has an alphabet of eight symbols: $\Gamma = \{\mathbf{0}, \mathbf{1}, \#, D, B, A, L, R\}$, although tighter bounds are achievable (*e.g.,* symbols **0** and **1** can be used instead of L and R).

Next, the UTM3 can be modified to simulate on-line DTMs with an arbitrary alphabet by encoding the alphabet and adjusting the parts of the machine that use it. For example, our three symbols could be encoded as follows: $\mathbf{0} = GA$, $\mathbf{1} = GAA$, $\# = GAAA$. Any finite alphabet can be encoded in this way.

From the Property 1 and the fact the the UTM3 is capable of simulating any DTM3, it follows that the computational power of the UTM3 is no less than the computational power of regular DTMs or UTMs. Now, we will prove the upper bounds of the space and time complexity for the UTM3 as compared to the DTM3 it simulates.

Property 2. *Let M_3 be a DTM3 with q internal states and space complexity $S_D(n)$ for input n. Then, the upper bound for space complexity $S_U(n)$ of the UTM3 simulating M_3 on input n is:*

$$S_D(n) + 3.5q^2 + 3.5q - 2. \tag{1}$$

Proof. Tapes T_{IN} and T_{OUT} act like streams and are not included in either $S_D(n)$ or $S_U(N)$. Thus, $S_D(n)$ is affected only by T_{DATA}, while $S_U(n)$ is also affected by T_{PROG} and T_{STAT}. Since the UTM3 simulates M_3 perfectly, then the space used by T_{DATA} for the UTM3 can never exceed the space of T_{DATA} used for M_3, which for input n equals $S_D(n)$. Next, T_{STAT} is only used for holding the intended state number in a unary format. Thus, if M_3 has q states, then the maximal space needed on this tape equals $q-1$ spaces for storing the state number (the first state is 0, so the last state is $q-1$) and one space reached when we go past the number, so q in total. The last tape to consider is T_{PROG}. The last state $q-1$ is, by convention, the final state, so it is composed of symbols D, H, and $q-1$ symbols A, so $q+1$ symbols in total. Any other of the remaining $q-1$ states need one symbol D for the beginning of the state (so $q-1$ symbols in total). After that, each state holds its number in a unary format. Since this concerns states from 0 to $q-2$, then the total space needed for that is the sum of integers from 0 to $q-2$ and equals $\frac{q^2-3q+2}{2}$. In the worst-case scenario, each non-final state is a normal state; therefore, it has three subsections. Each subsection holds the symbol to write (one space) and shift direction (one space). Moreover, subsections hold the target state designation, which (once again) can take up to q spaces. Thus, each subsection can take up to $q+2$ spaces; therefore, $3q+6$ for all subsections and $(q-1)(3q+6) = 3q^2+3q-6$ for all non-final states. Lastly, we need two blank symbols at the end of the tape whenever we reach the end of the program when searching for the next state candidate. This gives us a following total formula:

$$S_D(n) + q + (q+1) + \left(\frac{q^2-3q+2}{2}\right) + (3q^2+3q-6) + 2 =$$
$$= S_D(n) + 3.5q^2 + 3.5q - 2.$$

Since this is the worst case, the actual value may be lower in practice (which ends the proof). □

While proving the upper bound for the space complexity is relatively easy, the time complexity is more difficult; so, it is best to split this into three parts. First, let us consider the moment when the UTM3 has to switch to the next internal state of the DTM3 it simulates.

Property 3. *Let M_3 be a DTM3 with q internal states. Then, the upper bound for the number of steps k needed by the UTM3 simulating the M_3 to reach the next intended state is:*

$$10q^2 + 18q + 2. \tag{2}$$

Proof. As stated before, when the state reaching procedure is started, H_{PROG} is always placed directly to the left of the intended state designation (*e.g.,* BAA). Thus, two shifts to the right are needed to position H_{PROG} above the first A. Let us assume that T_{STAT} is cleared and that its head is positioned at the beginning. Since M_3 has q states and states are numbered from 0, then the longest state designation has

$q-1$ A's. Thus, $q-1$ steps are needed to copy the A's to T_{STAT}. Next, we need to reach the next candidate state and check it. If we don't reach the end of the program, then the next candidate will be found in $4q+6$ steps, because the beginnings of any two adjacent states cannot be further apart than this. If we do reach the end of the tape, we will need two extra shifts to realize this (reaching two blanks past the end of the program) and then up to another $4q+6$ steps to find a candidate state in the reverse direction. To sum it up, the next candidate (symbol D) can always be reached in $2(4q+6)+2 = 8q+14$ steps. Then, we shift once to position H_{PROG} above the first symbol A of the candidate state, and we use at most $q+1$ steps to move H_{STAT} to the beginning of its tape. Next, we need at most q steps to verify whether the candidate state is the intended one. After that, either the state is correct (and H_{PROG} will be placed to determine the state type) and the procedure ends, or the state is not correct and the next candidate state is sought. To sum it up, searching and checking a single state candidate takes up to $8q+14+1+(q+1)+q = 10q+16$ steps. In the worst-case scenario, we will need to check all of the states before we find the correct one; so, this value must be multiplied by q, giving $10q^2+16q$. Finally, we must clear T_{STAT}, because we assumed that it is clear before each procedure. Since after a successful check, H_{STAT} is placed past the end of its tape, then we need at most $q+1$ steps to clear it. This ends the procedure, giving us a total, worst-case-scenario number of steps equal to:

$$2+(q-1)+10q^2+16q+(q+1) = 10q^2+18q+2,$$

which ends the proof. □

Using the above upper bound for the intended state search procedure, we can provide the upper bound on the number of steps the UTM3 needs to complete a single step of the simulated DTM3.

Property 4. *The upper bound for the number of steps k needed by the UTM3 to simulate a single step of the DTM3 M_3 with q internal states is:*

$$10q^2+20q+8. \tag{3}$$

Proof. Let us assume that at the beginning of each step H_{PROG} is placed directly to the right of the current state designation; *i.e.,* directly above the type of the state indicated by symbols H, I, O, or L/R. If the symbol is H, the machine halts in a single step. If the symbol is I or O, then the machine inputs/outputs in one step and then starts the state-reaching procedure which, following the Property 3, completes in at most $10q^2+18q+2$ steps, for $10q^2+18q+3$ steps in total. The last possible case is a normal state. First, we need to position H_{PROG} at the corresponding subsection. The worst case is the blank subsection. To shift to it, we have to go past the first two subsections. As stated in Property 2, a subsection size is no greater than $q+2$; thus, we have to shift at most $2q+4$ times to reach the needed subsection. After this, we use two steps to 1) overwrite the current symbol on T_{DATA}, and shift T_{PROG} to the right, and 2) shift T_{DATA} according to the current value on T_{PROG}. After this, we start the state-searching procedure, which once again completes in $10q^2 + 18q + 2$ steps, for $10q^2+20q+8$ steps in total. Finally, we conclude that $10q^2+20q+8 > 10q^2+18q+1$, which ends the proof. □

With Properties 3 and 4, we can now present a proof for the upper bound on the number of steps needed by the UTM3 to fully simulate its DTM3 (assuming that the DTM3 halts on the given input).

Property 5. *Let M_3 be a DTM3 with q internal states and time complexity $T_D(n)$ for input n. Then, the upper bound for time complexity $T_U(n)$ of the UTM3 simulating M_3 on input n is:*

$$T_D(n)(10q^2 + 20q + 8) + 1. \tag{4}$$

Proof. The proof is almost entirely self-evident from the Property 4. If a single step of M_3 is simulated in at most $10q^2 + 20q + 8$ steps of the UTM3, then $T_D(n)$ steps of M_3 are simulated in at most $T_D(n)(10q^2 + 20q + 8)$ steps of the UTM3. The only thing missing is that, at the beginning, H_{PROG} is placed at the symbol D of the 0th state, and we need one shift to the right to place it in the correct position for the simulation of the first step. Thus, the final value is $T_D(n)(10q^2 + 20q + 8) + 1$. □

To sum up this subsection, we present two theorems concerning time and space complexity for the UTM3 model that are derived directly from the Properties 2 and 5.

Theorem 1. *Let $T_U(n, q)$ be the time complexity for the UTM3 that simulates the DTM3 with q states on input n and $T_D(n)$ be the time complexity of that DTM3 on input n. Then:*

$$T_U(n, q) \in O(q^2 T_D(n)). \tag{5}$$

Theorem 2. *Let $S_U(n, q)$ be the space complexity for the UTM3 that simulates the DTM3 with q states on input n and $S_D(n)$ be the space complexity of that DTM3 on input n. Then:*

$$S_U(n, q) \in O(S_D(n) + q^2). \tag{6}$$

From Theorems 1 and 2, we conclude that, when the UTM3 model of computation simulates a DTM3, it does not impose time or space complexity penalties that are dependent on the input n or (by extension) the size of that input. Thus, the increase in complexity is dependent only on the squared number of internal states of the DTM3 being simulated (therefore, it is also dependent on the problem being solved).

The upper bounds defined in Properties 2 and 5 apply to the worst-case scenario. The average state number for q states is $\frac{1}{2}q$. This means that the expected value of the number of steps and space needed should be reduced to $\frac{1}{4}$ of the worst-case upper bound (since $(\frac{1}{2}q)^2 = \frac{1}{4}q^2$). This was researched and proven using a simple computer simulation. Two computable functions *double* ($f(x) = 2x$) and *mod5* (outputs **1** if $x \mod 5 \equiv 0$ and **0** otherwise) were computed for natural numbers $x = \{1, \ldots, 50\}$ using both a standalone DTM3 and the UTM3 that simulated it. The required values of $T_D(n)$ and $T_U(n)$ (indicating number of steps for both machines) were measured and used to create a ratio $\frac{T_U(n)}{T_D(n)}$ representing how much slower the UTM3 is compared to the DTM3 it simulates. The upper bounds were computed using the expression from Inequality 4 with $q = 11$ (*mod5*) and $q = 13$ (*double*). The results are shown in Figure 4. We indeed observe that the average ratio is about $\frac{1}{4}$ of the theoretical upper bound.

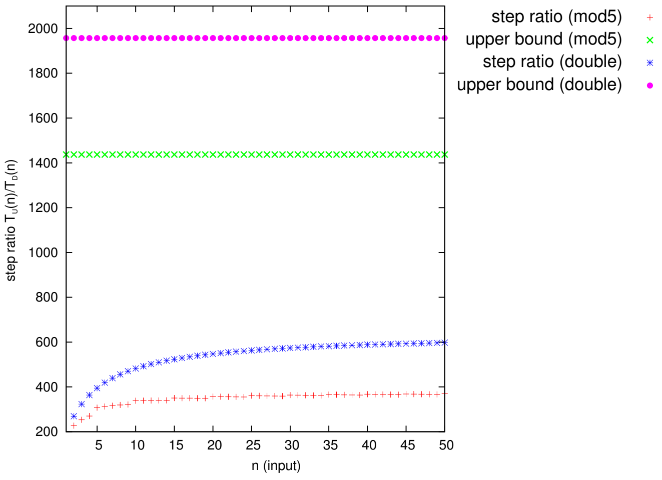

Figure 4. Comparison of theoretical upper bound and practical values of step ratio $\frac{T_U(n)}{T_D(n)}$ for two exemplary computable functions.

3. Dynamic computation model

The UTM3 model defined above is our own approach to the concept of the Universal Turing Machine; therefore, on its own it provides no new properties. However, we will now use this model and extend it to define a completely new model of computation, one able to support and apply runtime code changes for any DTM3. This model will be called the Dynamic UTM3, or simply the DUTM3.

3.1. Subtasks and repositions

Here, we will briefly and more formally describe the concept of subtasks introduced in [14] as well as how we will use it to support runtime code changes. A *subtask* is essentially a non-empty subset of states of a DTM (DTM3). A few additional conditions might be necessary, depending on the specific model. For example, subtasks should be disconnected; *i.e.,* no state should be a part of more than one subtask. This condition can be alleviated when we allow for parent-children subtasks; *i.e.,* a subtask that is a subset of another subtask. The second possible condition is that the subtasks should form a weakly connected digraph (graphs for Turing Machine transition functions are always directed graphs). One exception could be a subtask that consists of all states not officially assigned to any subtask. A simple example of a DTM3 with defined subtasks is shown in Figure 5, with the subtasks taking the form of rectangles.

Each subtask has a subset of states that can be used to enter a given subtask. These states are called *entries* of that subtask. State e is an entry of a subtask S if and only if there exists a state a from a subtask other than S and edge (a, e) exists. If a subtask is not isolated (*i.e.,* it can be accessed), then it contains at least one entry. Entries are labeled with the letter e in Figure 5. Let us notice that subtask 2 has multiple entries.

When a subtask entry is reached, the subtask is started, and it either continues to run forever, halts the DTM3, or completes (another subtask is started). Also, let us note that, at any given time (step), at most one subtask can execute (except parent-children subtasks). This is in contrast to, for example, recursive procedures known from programming languages like `C/C++`, where multiple procedures (functions) may execute at a time via structures like the call stack.

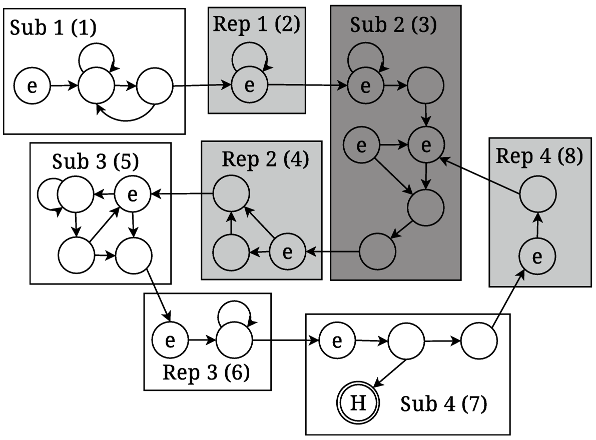

Figure 5. Example of a DTM3 state diagram with subtasks, repositions, and entries.

Let us notice that subtask "Rep 1" in Figure 5 is composed of only one state. Moreover, the state loops back into itself. This effectively means that this subtask can be "started" multiple times (essentially every step of the DTM3) before it completes. This will cause performance problems for our DUTM3, but the situation can be alleviated by changing this subtask into a form similar to "Rep 3." In this case, the entry is reached only once, regardless of the number of steps spent in the loop. This increases the number of states but will improve performance. This loop modification can be done automatically.

Because of what was stated above, each subtask is started in some context (contents of all tapes and positions of all tape heads) and then performs some processing until it completes. Therefore, each subtask is always tied to some computable function (algorithm) it computes. That means that each subtask effectively defines a (sub)DTM or, more precisely, a (sub)DTM3.

The main point and purpose of introducing subtasks is the ability to change some of the subtasks with limited or no effect on some other subtasks. In general, subtasks can be defined freely; however, for our approach to work, we impose one rule that is used when dividing (decomposing) the original DTM3 into subtasks. The rule states that the contents of the T_{DATA} must follow the so-called *expected format*. In essence, this means that each subtask has input and output protocols that describe the expected contents of the tapes before the subtask starts and after it completes. The specific definition of the expected format is left to the DTM3 designer (programmer).

Let us take the basic *quick sort* procedure as an example. We can divide quick sort into four subtasks: 1) Choose the pivot element; 2) move all elements with less value than the pivot before it and all elements with greater value after it; 3) sort the elements before the pivot; and 4) sort the elements after the pivot. Let us assume that steps 3 and 4 use unknown sorting algorithms (especially when the lists to sort are small enough to use naïve sorting). With this, the expected format rule between steps 3 and 4 is met: after step 3 is completed, the first half of the list is certainly sorted. If we tried to divide step 3 into two subtasks (3a and 3b), then it would be difficult since we do not know what the result would look like in the middle of subtask 3. Therefore, the choice of subtasks and their size is left to the designer.

The same rule also includes the fact that the tape heads must be placed in the correct positions for the subtask to complete successfully. For this, a special class of subtasks called repositions is used (indicated by “Rep” instead of “Sub” in Figure 5). Reposition is used to move the tape heads between subtasks and is not allowed to modify the tape contents (*i.e.,* it writes the same symbol it read from the tape before shifting). This effectively means that repositions always compute identity function $f(x) = x$. A given non-reposition subtask can have multiple repositions preceding and following it.

3.2. The DUTM3 model

Now, let us suppose that subtask “Sub 2” (dark gray in Figure 5) needs to change its algorithm (*i.e.,* it still computes the same function, but in different way). This means that repositions “bound” to this subtask have to be altered as well (light gray in the picture). This is the result of maintaining the cohesion of the DTM3, if “Rep 1”, “Rep 2” and “Rep 4” are not altered correctly, then “Sub 2” might fail to compute its function. Therefore, the original subtask change defines a wider range of subtasks that need to be altered. Finally, we obtain a list of subtasks to change. The size of the list depends on the original subtasks to change and the specific subtask graph of the DTM3. In our case, this list contains “Rep 1”, “Sub 2”, “Rep 2” and “Rep 4”. Now, we would like to develop a model of computation that will safely change these subtasks at runtime.

The first condition we assume is that the change can be performed safely only when the subtasks from the list are not executing. Therefore, the list is termed the *forbidden subtasks list*. Next, we assume that the subtasks are indexed with natural numbers starting from 1 (numbers in the parentheses in Figure 5). Thus, our forbidden list consists of subtasks 2, 3, 4, and 8.

Our next assumption is that the DUTM3 in its basic form will not question the validity of the changes supplied to it. Change authorization could be easily added, but even then the DUTM3 would not check the validity – as long as the changes are authorized, the DUTM3 will apply them when ready. This means that the responsibility for designing a valid change is left to the designer (*i.e.,* programmers and automation tools such as compilers). We are, therefore, only concerned with applying valid changes effectively and safely.

The DUTM3 will need to be supplied with changes; therefore, a new tape is needed. This tape will be called the *change tape* (T_{CH}, H_{CH}). This tape, unlike T_{IN} and T_{OUT}, is not fully a stream, because the DUTM3 will need to move back and forth along it. Thus, T_{CH} will be treated as a normal tape and be included in the space complexity of the DUTM3.

The changes supplied in the tape must follow a certain format. First is a forbidden list of subtasks that cannot execute for the change to occur. For this, we will use a new alphabet symbol C and encode subtasks numbers in a unary format, as with the state numbers. Therefore, our forbidden list of 2, 3, 4, and 8 would be encoded as:

CAACAAACAAAACAAAAAAAA

The changes to be made must be specified after the forbidden list. The most straightforward solution is to place the description of the entire new DTM3 on T_{CH}. Then, we can simply clear T_{PROG} and replace it with the description on T_{CH} when we are ready to perform the change. With this, we define the final format of the changes:

1. Special symbol S indicating the beginning of the change description.
2. Forbidden states list.
3. Blank symbol.
4. New DTM3 program (with the adjustment described below).
5. Symbol S indicating the end of the change description.

If more than one change description will be needed, then the next one will be placed after the previous one, separated by the blank symbol. For now, we ignore the issue of how and when the change descriptions appear on T_{CH}.

Let us notice that, for increasing the performance, the checks on whether the changes can be applied or not should be done only when a new subtask is reached. To this end, we need to mark the entries of each subtask by adding a state number for each entry in the form of CA, CAA, etc. as shown in Figure 6. All non-entry states remain as before.

The method of simulating a single step in the DUTM3 changes considerably compared to the UTM3. At the beginning of the step, H_{PROG} is placed directly past the number of the current state. In the UTM3, this would mean that the head was over symbol I, O, H, **0**, or **1** (the last two start the zero section for normal states). For the DUTM3, this is true only for non-entry states. At this point, the entry states will have symbol C, which starts the designation of subtask that was just entered. Thus, for non-entry states, we proceed exactly as for the UTM3. If there is new subtask, we check the current symbol on T_{CH}. If the symbol is blank, then there are no pending changes to apply. Thus, we shift T_{PROG} to the right until we go past the subtask number. Then, we are placed at the state indicator (symbols I, O, H, **0**, or **1**) and proceed exactly as for the UTM3.

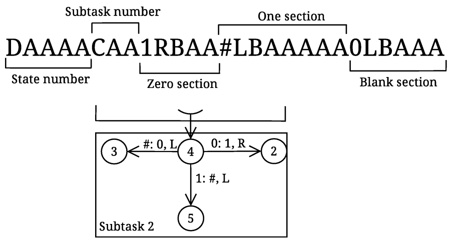

Figure 6. Example of a state description encoding for the DUTM3.

If the current symbol on the T_{CH} was S, however, then there are pending changes on this tape, and we will try to apply them. First, we need to check whether our current subtask is on the forbidden list. The check will commence similarly to how we checked whether the candidate state was the correct one. We start by copying the entered subtask to one of the tapes. T_{STAT} is currently free, so we can use it without attaching any additional tapes to the DUTM3. Thus, we shift H_{PROG} to the right so it is placed on the first A of the subtask number, and we copy it to the T_{STAT}. Next, we shift H_{STAT} to the beginning and shift H_{CH} one space to the right.

Now, both T_{STAT} and T_{CH} are placed above the first A of the current/forbidden subtask to check. We then commence the check. If we run out of A's on T_{CH} at exactly the same moment that we run out of them on T_{STAT}, then this means that the current subtask is indeed on the forbidden list. Thus, we shift T_{CH} to the left until we reach the S symbol, restoring the head to the position before the check, and we clear T_{STAT}. After this, we continue as the regular UTM3 would (T_{PROG} is placed past the subtask designation – exactly where we will need it).

If we ran out of A's on T_{CH} or T_{STAT} before the other tape, then the current subtask does not match the one on the forbidden list; but, the list may have another forbidden subtask. Therefore, we shift H_{STAT} to the beginning of its tape, and we shift H_{CH} past the A symbols (if it is needed). After that, H_{CH} will be placed either over symbol C or #. If the symbol is C, then there are more subtasks on the list, and we recommence the check immediately. If the symbol is #, however, then we have reached the end of the forbidden list, and our current subtask was not on the list. We shift H_{CH} once to the right, so it is placed past the blank and above the first symbol of the new program – the changes can be applied now.

In order to apply a change, first we need to store the current state so the DUTM3 can resume its simulation after the change was performed. We start by clearing T_{STAT}, which still holds the current subtask number. Thus, we shift H_{STAT} to the right, clearing the symbols on our way, and then we shift back to the beginning. Next, we shift T_{PROG} back to the left until we reach symbol D, meaning we are back at the state number. Now, we copy this number to T_{STAT}. Next, we clear the old program from T_{PROG} simply by shifting H_{PROG} to the end of the tape and then shifting back to the beginning, replacing all symbols with blanks.

After the program has been cleared, we simply copy the new program from T_{CH} to T_{PROG}. This process ends when we reach symbol S on T_{CH}. Then, we shift H_{CH} twice to the right. The first shift will place the head on the blank symbol in between the change descriptions. The second shift will place the head at the beginning of the next change description (if any). All that is left is to switch the DUTM3 to the state it was in when performing the change. We do this by starting the already-known candidate state search procedure with one exception – the intended state is already on T_{STAT}, so we just start by searching for the next candidate. Moreover, H_{PROG} is conveniently placed past the program, so the procedure will quickly detect the end of the program and will reverse the direction search, possibly reducing the average time needed to find the intended state.

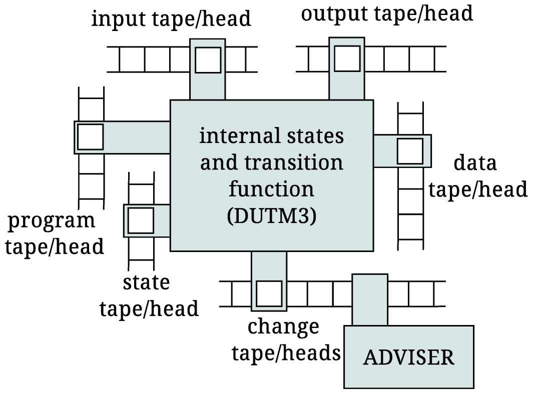

Figure 7. The structure of the 3-symbol on-line Dynamic Universal Turing Machine (DUTM3).

Our last concern is how the contents of T_{CH} are created. We assume that this tape is shared by two machines. One machine is the DUTM3, and it can only read the tape. After the change has been applied, the DUTM3 simply shifts H_{CH} to the place where next change description is supposed to be. The other machine will be called the *adviser* and can only write to the tape. The adviser can decide to write to T_{CH} at any time. When it decides to do so, it writes down the entire description and then leaves one blank symbol between descriptions, so any subsequent descriptions will appear further on the tape. This means that the adviser has its own head to modify the tape. We can also assume that the adviser works at the same speed as the DUTM3. The DUTM3 will never proceed faster than a space per step (especially since forbidden-list checking takes many more steps), so the DUTM3 will never outrun the adviser and read the parts of the description that the adviser has yet to write. We also assume that the strategy of the adviser is generally unknown and may change, even if the DUTM3 simulates the same DTM3. Moreover, the adviser can be thought of as another DTM3, with T_{CH} being its output tape. The final structure of the DUTM3 model of computation is shown in Figure 7. Let us notice that this DUTM3 has a tape alphabet of ten symbols: $\Gamma = \{\mathbf{0}, \mathbf{1}, \#, D, B, A, L, R, C, S\}$.

As a final note for this section, let us mention that our concept of the adviser and its effect on the main DTM is different from the concept of advice known from the literature. The original notion of advice (used by Arora and Barak [1], among others) is described as an additional input to a Turing Machine and is allowed to depend on the size of the input. Thus, for input x, the advice string is $A = f(n)$, where $n = |x|$. This notion of advice was used to define several complexity classes. In general, a decision problem is described as being in class P/f(n) if there is a polynomial Turing Machine M and advice A of length $f(n)$ such that, for each input x, machine M correctly decides the problem, given x and A.

While the above notion of advice has some similarities with our notion of the adviser, these two concepts have fundamental differences that can be summed up as follows:

- In our approach, the central Turing Machine has no restriction of being polynomial. It is true, however, that polynomial Turing Machines are desirable in general.
- It is possible for the adviser to produce "incorrect" advice, produce no advice at all or produce advice more than once. It is even possible that a different (but still "correct") advice will be produced while running the Turing Machine twice for the same input x. This is completely different from the advice described by Arora and Barak, where the advice has to be constant for a given x.
- Our adviser approach is focused on providing the Turing Machine with dynamic runtime change properties, while the original advice is treated as just additional input and the entire Turing Machine remains static.

4. Properties

In this section, we will study several properties of the DUTM3, using the previously researched information about the UTM3, including computer simulations and theoretical properties. Some parallels between the DUTM3 model and Oracle Machines will also be drawn.

4.1. Runtime code changes

The initial research concerns the runtime-code-change mechanism itself. In particular, we would like to show that such changes are indeed applied at runtime and that they have the intended effect. Since Turing Machines can become very complex (even for common problems like sorting), we will restrict ourselves to very simple examples.

Assume we have a server that doubles every number it receives, called *Double-Server*. In other words, we designed an on-line DTM3 that computes $f(\bar{x}) = 2\bar{x}$ for a vector of inputs $\bar{x}$. Therefore, for input $\bar{x} = [2, 4]$, the output would be $[4, 8]$. In our DTM3, we represent numbers in unary, with input numbers separated by blanks and ending with **0**. Input $[2, 4]$ on T_{IN} would, therefore, be **11#1111#0**. Finally, let the output numbers be concatenated without using blanks (we will use this property in a moment); therefore, T_{OUT} for the aforementioned input vector would simply be the number 12 $(4 + 8)$ in unary. This DTM3 has 12 states and consists of 4 subtasks: input, double, output, and one reposition (the other repositions are simply empty).

Now, let us consider performing two runtime changes. The first change assumes that the original function $f(\bar{x}) = 2\bar{x}$ was incorrect (not intended) and we would now like to triple numbers instead of doubling them. The output without separating blanks gives us a way to measure the correctness of our program. The closer the answer is to $3\bar{x}$, the better. If the answer is exactly $3\bar{x}$, then all numbers were tripled correctly (100% success). On the other side, if the answer is exactly $2\bar{x}$, then all numbers were outputted incorrectly (0% success).

Thus, we designed a new version of this DTM3 with one additional state. The forbidden list contains subtasks 2 and 3. Research was conducted for inputs with a sum of 50 (thus $2\bar{x}$ yields unary 100 and $3\bar{x}$ yields unary 150). The runtime change was performed when a certain number of steps have passed (the adviser placed the change description on T_{CH} at the designated time). The results are shown in Table 1. Steps counted are DUTM3 steps and not the steps of the DTM3 being simulated.

Table 1

Results of the error-correcting runtime change.

Change step	Output	Change step	Output
1	150	150 000	120
10 000	148	170 000	116
30 000	144	190 000	112
50 000	140	210 000	108
70 000	136	230 000	104
90 000	132	250 000	100
110 000	128	270 000	100
130 000	124	no change	100

From the table, we see that applying the change as soon as possible (step 1) allowed us to receive the best possible result (output value 150). This is due to subtask 1 not being on the forbidden list and the DTM3 program being changed before the first doubling subtask was ever started. We also see that the later the change was applied, the worse the result. Above about 250,000 steps, it is too late to perform the change (all computations are already done), so the output is the same as for no changes at all. This proves that the DUTM3 model of computation allows us to correct errors at runtime to some extent.

The second kind of change concerns performance. The original *DoubleServer* doubles the number and then outputs each **1** to T_{OUT}. This is redundant – it is enough to output each **1** twice as soon as the input is read. Thus, we completely remove the output subtask and modify the multiplying subtask to output the data right away (this can be done even faster, but not with the subtasks we defined). The new DTM3 has only nine states, and the forbidden list now contains subtasks 2, 3, and 4. The research was done for ten instances, the input of each instance consisted of the number x repeated a number of times (the same number x was used everywhere for repeatability). The change description was placed in all cases on T_{CH} after 500,000 steps had passed. These results are shown in Table 2. Step ratio is simply the number of steps for the DUTM3 divided by the number of steps for the UTM3.

Table 2
Results of the performance-increasing runtime change.

# of inputs	UTM3 steps	DUTM3 steps	Step ratio
1	654 690	670 987	1.025
2	1 308 998	709 222	0.543
3	1 963 306	747 457	0.382
4	2 617 614	785 692	0.300
5	3 271 922	823 927	0.252
6	3 926 230	862 162	0.220
7	4 580 538	900 397	0.197
8	5 234 846	938 632	0.179
9	5 889 154	976 867	0.166
10	6 543 462	1 015 102	0.155

For the first input, the DUTM3 works slower than the UTM3. This is because it had only one input number, and at 500,000 steps, it was too late for the changes to take effect. For all other cases, the DUTM3 works considerably faster, even though it is generally slower than the UTM3 (as we will prove later on) and wasted some time applying the change. For the last case, the DUTM3 was more than six times faster than the UTM3. Of course, this value is thanks to the reduced time complexity of the doubling algorithm after the change, but the DUTM3 was what allowed us to take advantage of this at runtime. We conclude that the DUTM3 model of computation allows us to affect the properties (like the time and space complexity) of the algorithms used at runtime.

4.2. Time and space complexity

In this subsection, we will study the theoretical properties of the DUTM3 (like its time and space complexity). Since the DUTM3 is a new model of computation (unlike the

UTM3), we will establish these properties as theorems. Let us start with the space complexity of the DUTM3 when no changes are assumed.

Theorem 3. *Let M_3 be a DTM3 with q internal states, s subtasks, and space complexity $S_D(n)$ for input n. Then, the upper bound for space complexity $S_{DU}(n)$ of the DUTM3 simulating M_3 on input n is:*

$$S_D(n) + 3.5q^2 + (4.5 + s)q. \tag{7}$$

Proof. From Property 2, the space complexity for the UTM3 is $S_D(n)+3.5q^2+3.5q-2$. Now, we simply add the space complexity that the DUTM3 itself adds. T_{DATA} for the DUTM3 works exactly as for the UTM3. T_{PROG}, however, needs additional space to hold the current subtask for the entry states. In the worst-case scenario, each state in M_3 can be an entry for the last subtask (whose number is equal to s), thus requiring $s + 1$ spaces. Therefore, all q states can require $q(s + 1)$ additional space on T_{PROG}. Now, let us consider T_{STAT}, which is responsible for holding the subtask number or state number in the DUTM3 (but only one at a time). The maximal subtask number is s, thus requiring at most s spaces in unary to hold it and one more space for the blank symbol at the end (for $s + 1$ spaces in total). Since s cannot be larger than q (each state can belong to at most one subtask), then this value can be reduced to $q + 1$. Since the UTM3 needed at most q spaces on T_{STAT}, then this is now increased by one. As for T_{CH}, we assumed no runtime changes; but the DUTM3 will still scan T_{CH} to check for them, so at least one space for the blank symbol is needed. To sum it up, we require at most an additional $q(s + 1) + 2$ spaces compared to the UTM3, so $S_D(n) + 3.5q^2 + (4.5 + s)q$ in total. □

Now, we would like to make upper bounds for the space complexity with changes included. However, it is difficult to consider both the space of the current program and the space of the new program at the same time, since both programs can have a quite different number of states and subtasks. The resulting space complexity would be dependent on many variables; thus, we decided to simply consider the space complexity needed on T_{CH} to store a series of changes.

Theorem 4. *Let $\bar{C} = \{C_1, C_2, \ldots, C_m\}$ be a series of m runtime changes (change descriptions) to be applied to a DTM3 M_3. Change C_i has q_i internal states and s_i subtasks. Then, the upper bound for the space $S(\bar{C})$ needed for $\bar{C}$ on T_{CH} is:*

$$m(3.5q_{max}^2 + (3.5 + s_{max})q_{max} + 0.5s_{max}^2 + 1.5s_{max} - 2) - 1, \tag{8}$$

where $q_{max} = \max_i q_i$ and $s_{max} = \max_i s_i$.

Proof. Let us consider description size for change C_i. From Theorem 3, we know that the space needed for storing the program of a DTM3 with q states and s subtasks takes up to

$$3.5q^2 + 2.5q - 4 + (s + 1)q = 3.5q^2 + (3.5 + s)q - 4$$

spaces (minus the two blanks from T_{PROG} needed to detect the program's end). Next, we need space to store the forbidden list. The list can contain at most $s-1$ subtasks (at least one subtask cannot be forbidden or the change will never be applied); therefore, in the worst-case scenario, the list will contain subtasks from 2 to s. Subtask i needs i spaces plus one space for symbol C, so $i+1$ spaces. Thus, the forbidden list will need at most $\sum_{i=2}^{s}(i+1) = \frac{1}{2}(s^2+3s-4)$ spaces. Next, we need one blank symbol between the forbidden list and the program as well as one S symbol at the beginning and the end of the description. Finally, the space needed for the description of change C_i is at most:

$$3.5q_i^2 + (3.5+s_i)q_i - 4 + \frac{1}{2}(s_i^2+3s_i-4) + 3 = \\ = 3.5q_i^2 + (3.5+s_i)q_i + 0.5s_i^2 + 1.5s_i - 3$$

spaces. From this, we apply the above formula for the biggest change in $\bar{C}$: $C_{\max}$, with $q = q_{\max}$ and $s = s_{\max}$. No other change in $\bar{C}$ will need more spaces; thus, the final space needed for all m descriptions is at most:

$$m(3.5q_{\max}^2 + (3.5+s_{\max})q_{\max} + 0.5s_{\max}^2 + 1.5s_{\max} - 3) + m - 1 = \\ = m(3.5q_{\max}^2 + (3.5+s_{\max})q_{\max} + 0.5s_{\max}^2 + 1.5s_{\max} - 2) - 1.$$

The $m-1$ part comes from the fact, that we need a blank symbol between each pair of adjacent descriptions (so, $m-1$ blanks in total). □

Next, we consider the time complexity. We start with the basic complexity of the DUTM3 without assuming any changes. This will serve as a comparison to the time complexity of the UTM3 researched in Property 5.

Theorem 5. *Let M_3 be a DTM3 with q internal states, s subtasks, and time complexity $T_D(n)$ for input n. Then, the upper bound for time complexity $T_{DU}(n)$ of the DUTM3 simulating M_3 on input n with no changes is:*

$$T_D(n)(10q^2 + (2s+22)q + s + 9) + 1. \tag{9}$$

Proof. From Property 3, the beginnings of any two adjacent states of the UTM3 are not further apart than $4q+6$ spaces on T_{PROG}. On the DUTM3, a state can have an additional $s+1$ spaces for the subtask number. Then, if follows that the beginnings of two adjacent states for the DUTM3 are not further apart than $4q+s+7$ spaces. Therefore, in the DUTM3, the next state candidate can always be reached in $2(4q+s+7)+2 = 8q+2s+16$ steps. Next, we add the candidate-check procedure, and we get $10q+2s+18$ steps in total. Now, this procedure will happen at most q times (check all of the candidates before we find the correct state) for $10q^2+(2s+18)q$ steps. As in Property 3, we add 2 (shift before we copy state), $q-1$ (copy intended state to T_{STAT}), and $q+1$ (clear T_{STAT}) steps. Consequently, reaching the next state for the DUTM3 will take at most:

$$10q^2 + (2s+18)q + (q+1) + (q-1) + 2 = 10q^2 + (2s+20)q + 2.$$

Now, the UTM3 was adding $2q+4$ steps to omit subsections for zero and one (in the worst case) and then added 2 to overwrite and shift H_{DATA}. The only difference for the DUTM3 is that we might also need to shift over the subtask number before we reach the subsections, adding another $s+1$ steps (so $2q+s+7$ in total). Therefore, a single step of the DTM3 is simulated in:

$$10q^2 + (2s+20)q + 2 + 2q + s + 7 = 10q^2 + (2s+22)q + s + 9$$

steps of the DUTM3. Thus, the number of steps needed by the DUTM3 to fully simulate M_3 is at most:

$$T_D(n)(10q^2 + (2s+22)q + s + 9) + 1.$$

□

The last of these four theorems concerns the upper bound for the number of steps needed to apply a given change. This case is the most complicated, as the current program and the change description might have a different number of states (and subtasks). This means that the required number of steps is dependent on more variables.

Theorem 6. *Let C_i be a runtime change (change description) to be applied to a DTM3 M_3. C_i has q_i internal states and s_i subtasks. M_3 has q_m internal states and s_m subtasks. Then, the upper bound for time $T(C)$ needed for C to be applied to the DUTM3 simulating M_3 is:*

$$2s_m^2 + 7q_m^2 + 13.5q_i^2 + 3s_m + (7 + 2s_m)q_m + (21.5 + s_i)q_i.$$

Proof. We start the change-applying procedure when the current symbol on T_{CH} is scanned and turns out to be S (first step). Then, we shift T_{PROG} to the right to place it above the first A symbol of the current subtask (second step). Now, copying of our current (for M_3) subtask number to T_{STAT} commences and will take at most s_m steps.

Checking each subtask on the forbidden list is composed of: 1) shifting H_{STAT} back to the beginning of its tape (at most s_m steps); 2) shifting H_{CH} once to the right (one step); and 3) checking whether the subtasks match (at most $\min\{s_i, s_m\} \leq s_m$ steps). Thus, checking each subtask on the forbidden list takes up to $2s_m + 1$ steps. The forbidden list can have at most $s_m - 1$ subtasks, so this process can happen at most $s_m - 1$ times. Thus, the forbidden list check will take at most $(s_m - 1)(2s_m + 1) = 2s_m^2 - s_m - 1$ steps and is independent from s_i.

If we passed the forbidden list, then H_{CH} is now placed over the blank symbol between the list and the new program, so we shift it once to the right (one step). Now, we have to store the current state number on T_{STAT}, and we start by clearing this tape (which still holds the subtask number). Since H_{STAT} can be anywhere between the tape's beginning and end, we need at most $2s_m$ steps to clear it (s_m to reach the end, and another s_m to shift to the beginning). Now, we need to shift H_{PROG} back to the state number. This means shifting back to the subtask number (one step), through subtask number and symbol C (s_m+1 steps) and then through state number (at most q_m steps).

Now, H_{PROG} is placed above the D symbol, so we shift once to the right and copy the current state number to T_{STAT} in at most $q_m - 1$ (states are numbered starting with zero) steps. Now, we have to clear the old program from T_{PROG}, and we do this by reaching the end of the program (indicated by the double blank symbol), clearing everything along the way, and then returning to the beginning of T_{PROG}. Thus, in the worst-case scenario, we have to travel the entire program on the tape twice. As we know, the program with q_m states and s_m subtasks can take at most $3.5q_m^2 + (3.5 + s_m)q_m - 2$ spaces (including the double blank), so clearing it will take at most $7q_m^2 + (7 + 2s_m)q_m - 4$ steps. Next, we copy the new program from T_{CH} to T_{PROG}. This program will take up at most $3.5q_i^2 + (3.5 + s_i)q_i - 4$ spaces (without double blanks), and copying will consequently take up the same number of steps.

Next, we shift H_{CH} twice to the right so it is placed correctly to scan further changes. Lastly, M_3 needs to reach the internal state it was before the change, and we achieve this by starting the next state-reaching procedure, which will take no more than $10q_i^2 + 18q_i + 2$ steps. Thus, the final value is:

$$2s_m^2 + 7q_m^2 + 13.5q_i^2 + 3s_m + (7 + 2s_m)q_m + (21.5 + s_i)q_i.$$

□

We have previously assumed that each subtask should have at least one internal state. From this, it is obvious that the number of subtasks is at most equal to the number of internal states ($s \le q$). With this remark, we can present less-complicated upper bounds:

Corollary 1. *The space complexity from Theorem 3 is:*

$$S_{DU}(n) \le S_D(n) + 4.5q^2 + 4.5q. \tag{10}$$

Corollary 2. *The space complexity from Theorem 4 is:*

$$S(\bar{C}) \le m(5q_{max}^2 + 5q_{max} - 2) - 1, \tag{11}$$

where $q_{max} = \max_i q_i$.

Corollary 3. *The time complexity from Theorem 5 is:*

$$T_{DU}(n) \le T_D(n)(12q^2 + 23q + 9) + 1. \tag{12}$$

Corollary 4. *The time complexity from Theorem 6 is:*

$$T(C) \le 11q_m^2 + 14.5q_i^2 + 10q_m + 21.5q_i. \tag{13}$$

If we additionally assume that the number of states for the new version is equal to the old version (*i.e.*, $q_m = q_i$), then the Inequality 13 can be reduced to $T(C) \leq 25.5q_m^2 + 31.5q_i$.

Now, we would like to make more-general conclusions on the time and space complexity for the DUTM3 (similar as with Theorems 1 and 2). The proofs follow immediately from Corollaries 1 and 3.

Theorem 7. *Let $T_{DU}(n,q,s)$ be the time complexity for the DUTM3 that simulates the DTM3 with q internal states and s subtasks on input n, and $T_D(n)$ be the time complexity of that DTM3 on input n. Then:*

$$T_{DU}(n,q,s) \in O((q^2 + sq + s)T_D(n)). \tag{14}$$

If we simplify with $s \leq q$ then

$$T_{DU}(n,q) \in O(q^2 T_D(n)). \tag{15}$$

Theorem 8. *Let $S_{DU}(n,q,s)$ be the space complexity for the DUTM3 that simulates the DTM3 with q internal states and s subtasks on input n, and $S_D(n)$ be the space complexity of that DTM3 on input n. Then:*

$$S_{DU}(n,q,s) \in O(S_D(n) + q^2 + sq). \tag{16}$$

If we simplify with $s \leq q$ then

$$S_{DU}(n,q) \in O(S_D(n) + q^2). \tag{17}$$

From these properties, we conclude that the DUTM3 has asymptotically the same time and space complexity as the UTM3 (when both simulate the same DTM3). However, the DUTM3 is slower than the UTM3. We can use the Inequalities 4 and 12 to compare the time complexities of the UTM3 and DUTM3; with this, we state the following theorem (which simply creates a rational function and then computes its limit, with q approaching positive infinity).

Theorem 9. *If $T_U(n)$ and $T_{DU}(n)$ are upper bounds for the time complexities of the UTM3 and DUTM3, respectively, simulating a given DTM3 on input n, then:*

$$\frac{T_{DU}(n)}{T_U(n)} = \frac{T_D(n)(12q^2 + 23q + 9) + 1}{T_D(n)(10q^2 + 20q + 8) + 1}, \tag{18}$$

$$\lim_{\{q, T_D(n)\} \to \{\infty, \infty\}} \frac{T_{DU}(n)}{T_U(n)} = \frac{6}{5}, \tag{19}$$

$$\sup \frac{T_{DU}(n)}{T_U(n)} = \frac{6}{5}, \tag{20}$$

$$\inf \frac{T_{DU}(n)}{T_U(n)} = \frac{15}{13}. \tag{21}$$

The function from Eq. 18 is monotonically non-decreasing on both variables (for positive integers). Moreover, the number of states q and execution time $T_D(n)$ cannot be lower than 1, and the limit is as stated in Eq. 19. From this, we conclude that the function is bounded as shown in Eq. 20 and 21. The infimum is computed based on the smallest element *i.e.,* $T = q = 1$. Thus, the execution time of the DUTM3 is no more than 20% greater than the upper bound for the UTM3.

This 20% seems significant, but it is only an upper bound (and in practice, this value will be much lower). For example, let us consider the function of two natural numbers $f(a, b) = (2a + 4)/3 + 3(b - 3)/2$. A DTM3 for this function was designed, with 47 internal states and 15 subtasks. During testing (via computer simulation), it turned out that the DUTM3 simulating this DTM3 was only 3% slower than when the UTM3 simulated it. Moreover,let us notice that this DTM3 had one subtask every three internal states (close to the minimum of one state per subtask), which is very impractical. In real-life applications, such a computable function is but a single statement. Consequently, real-life subtasks would contain at least dozens of states, though values in the thousands are very likely as well. To sum it up, the DUTM3 model of computation has little influence on the time complexity (as long as new subtasks are not entered too often).

Now, we can establish a similar theorem for the space complexity using Inequalities 1 and 10.

Theorem 10. *If $S_U(n)$ and $S_{DU}(n)$ are upper bounds for the space complexities of the UTM3 and DUTM3 respectively, simulating a given DTM3 on input n then:*

$$\frac{S_{DU}(n)}{S_U(n)} = \frac{S_D(n) + 4.5q^2 + 4.5q}{S_D(n) + 3.5q^2 + 3.5q - 2}, \tag{22}$$

$$\lim_{q\to\infty} \frac{S_{DU}(n)}{S_U(n)} = \frac{9}{7}, \tag{23}$$

$$\lim_{S_{DU}(n)\to\infty} \frac{S_{DU}(n)}{S_U(n)} = 1, \tag{24}$$

$$\sup \frac{S_{DU}(n)}{S_U(n)} = \frac{5}{3}, \tag{25}$$

$$\inf \frac{S_{DU}(n)}{S_U(n)} = \frac{9}{7}. \tag{26}$$

The function from Eq. 22 is monotonically non-increasing; but unfortunately, a limit on either variable does not exist. We can still conclude that the largest element is when $S = q = 1$; thus, yielding our infimum. The supremum is based on the single-variable limits.In result, the DUTM3 simulating a given DTM3 will need 67% more tape space than the UTM3 (in the worst-case scenario).

Moreover, for practical values of S and q, this additional needed space quickly approaches $\frac{2}{7}$ (about 28.6%) or 0%, depending on the values of S and q. We conclude that the DUTM3 simulating a given DTM3 will need only about 28.6% more space than the UTM3 simulating the same DTM3 (as long as q and $S_{DU}(n)$ are large enough).

4.3. Oracle Machines and computational power

If we assume no changes done at runtime, then our DUTM3 is reduced to a slower version of the UTM3. Since the UTM3 is capable of simulating any DTM3, the DUTM3 is therefore capable of simulating any DTM3 as well. As proved in Property 1, a DTM3 exists for every computable function. Thus, the DUTM3 is capable of computing any computable function $f(x) = y$ as long as we encode input x and output y over alphabet $\{\mathbf{0}, \mathbf{1}, \#\}$ that is computed by Turing Machines. Let us denote this set of functions as R (functions μ-recursive).

This means that the computational power od the DUTM3 model is no lower than the computational power of a regular Turing Machine. Now, we would like to determine whether the computational power of DUTM3 is greater than that of a Turing Machine. In order to do this, let us consider a few possible classes of advisers.

4.3.1. Empty adviser

An empty adviser is one that does not generate any change descriptions whatsoever. In this case, the adviser does not affect the way the DUTM3 simulates its DTM3 and, consequently, has no effect on its computational power. Thus, the DUTM3 with an empty adviser has exactly the same computational power as the UTM3. The DUTM3 model with an empty adviser is physically possible to construct (except for the unbounded tape sizes).

4.3.2. Random adviser

A random adviser produces completely random (but valid) change descriptions. The number of steps between descriptions is random as well. We can assume that all change descriptions are equally likely or that the probability of a change description is inversely proportional to its size (space complexity). In both cases, any change description (including empty ones) is possible, so each DTM3 can receive a runtime change that will "break" it; *i.e.,* cause the DTM3 to produce incorrect output. In result, no DTM3 can guarantee that it will produce the correct answer with a random adviser, since the probability of receiving a "breaking" runtime change at any time is greater than zero. Thus, the computational power of the DUTM3 with a random adviser is no greater than for the regular UTM3. The DUTM3 model with a random adviser is physically possible to construct as well.

4.3.3. Oracle adviser

Let us modify the DUTM3 a little so it is capable of writing to T_{CH} as well (instead of merely reading it). Thanks to this, the DUTM3 can fully communicate with the adviser (of course, we can just add another tape that is writable by the DUTM3 and readable by the adviser). Thus, the DUTM3 can now directly post requests to the adviser. This is a difference, because the original DUTM3 had no way of influencing the decisions of its adviser.

Now, let us consider the halting problem. This problem is undecidable; *i.e.,* it cannot be solved (in general) by Turing Machines like the DTM3 or UTM3. However, DUTM3 can now use T_{CH} to "ask" specific "question". In particular, DUTM3 can "ask" the adviser by posting the data of the halting problem the DUTM3 is trying to solve. Now, let us also assume that the adviser is capable of solving that problem in a single step. In result, the adviser can create a runtime change ("answer") that will cause the DUTM3 to halt with the correct answer.

The outcome of the above modification is, in fact, a flavor of the Oracle Machine introduced by Turing in one of his papers (now a part of the papers collection by Martin Davis [5]). With this, our modified DUTM3 can now solve any problem solvable by Oracle Machines, as long as the adviser (now called the Oracle adviser) posts the appropriate runtime change. Since we currently have no means of creating Oracle Machines, we are not able to construct the DUTM3 model with an Oracle adviser either. Since this specific adviser transformed our DUTM3 into an Oracle Machine, then the concept of advisers can be thought of as a form of generalization of Oracle Machines.

4.3.4. Real-life advisers

The concept of adviser was originally introduced to model the behavior of the designer of the simulated DTM3. Thus, if DTM represents the computer program (software), then the adviser represents software developers (programmers). However, that also means that the properties of a real-life adviser are difficult to gauge. We can assume that such an adviser is neither empty nor an Oracle adviser. This leaves us with a random adviser, though its probability distribution remains unknown. This effectively means that the upper bound for the computational power of the DUTM3 with a real-life adviser is the same as for a regular UTM3.

5. Discussion

In this section, we will discuss the practical implications of the results obtained, including basic implementation possibilities. We start with a presentation of the key theoretical features of the presented approach.

The DUTM3 model is defined based on the notion of computation and therefore it is a general solution. More accurately, the DUTM3 does not assume any specific programming languages or paradigms, meaning that it is applicable for all Turing-complete languages and paradigms. This is a progress compared to the most of the practical solutions proposed in the literature, where a given solution is usually applied to only a specific language (usually C/C++ or Java), and the possibility of extending the solution to other languages is questionable at best. Moreover, some solutions are not only designed to allow changes to specific programming paradigms (usually object-oriented programming), but sometimes *require* other paradigms to work properly (like aspect-oriented programming or software agents). The DUTM3, on the other hand, does not rely on any such paradigms and, therefore, has the potential of being applied to a wide array of languages (including declarative/functional languages like Haskell). Actually, functional languages are closer to the original notion of computation than imperative ones, so implementing the DUTM3 model in these languages could prove easier. In result, the DUTM3 can be viewed as an entirely independent programming paradigm.

In Section 1, we mentioned the features of the dynamic programming languages. While such features are great tools that make runtime changes easier, they still differ fundamentally from the DUTM3. By using the graphs of subtasks, the DUTM3 allows for a convenient way of determining the range of a given dynamic change and when and how that change can be applied. The programmer only needs to define the subtasks, and these subtasks can correspond to an arbitrary piece of code (it can be an entire method or only some instructions). Moreover, it is possible to make the subtask definitions automatically, reducing the effort of the programmer. In comparison, using the Java reflection on its own would be problematic, as this would require the programmer to determine whether the insertion of a given class at a given moment is possible or not. In short, features like reflection mechanism can be used to implement the idea behind the DUTM3. The difference between the two is that the DUTM3 already provides the complete method for safe and coherent applying of properly defined dynamic changes, while reflections is just a low- level method for code inspection and modification with no inherent awareness of dynamic changes, their possibility of happening and requirements.

Finally, the DUTM3 model is uniform, meaning not only can it be applied to a wide array of languages, but it can also be applied to all of them with the same principle. This, in turn, means that the theoretical overhead and other properties should hold for all such cases. In practice, however, performance will also depend on the instruction set of the processor, the operating system, and the language type (compiled or interpreted languages).

Let us now proceed to a discussion about the practical implementation of the idea behind the DUTM3. The first problem is the fact that the DUTM3 model treats the computation the same way as a typical Turing Machine does – using a graph of states and the transition function. Thus, programs for the DUTM3 are vastly different than real-life software. In conclusion it would be easier to apply the DUTM3 with some intermediary step.

Fortunately, such a step exists – in the form of a Random Access Machines (RAM) model [4]. Such machines are equivalent to Turing Machines, with Random-access Stored-Program machines (RASP) serving as equivalents to Universal Turing Machines [7]. The benefits of using RAM/RASP models is that the structure and behavior of these models are similar to real computer systems; *i.e.,* the RASP model uses registers that correspond to the processor and memory registers. Moreover, RASP programs are defined as sequences of instructions and closely resemble programs written in typical assembler language, with instructions such as `ADD reg1 reg2`, etc.

Thus, we should try to extend the DUTM3 idea by defining it in the form of a RASP machine. Such an extension requires additional research and was outside the original scope of this paper; nonetheless, the research was carried out, and the result is the Dynamic RASP (DRASP) model proposed in [15]. With such a model, we can now propose a way to implement the DUTM3 idea in a real-life computer system.

First, we need some way to implement the adviser. While simply establishing a network connection between a given application and the website providing dynamic changes (similar to typical software update) is possible, we would like to try a slightly modified approach by introducing a proxy present on the system where the application is executed. This proxy could either be a part of the operating system (this requires changes to the OS, but can be more beneficial, because the OS is capable of directly altering the memory) or a dedicated standalone application. The proxy should be provided with information about the installed applications, their versions, and IP addresses for the update sites for each application. Thus, the proxy can query, download, and store the dynamic changes. It can also perform some additional operations, like uncompressing and decrypting the changes using well-known algorithms. This can improve performance and take care of some safety issues, providing some guarantee that the obtained dynamic changes were not tampered with.

Next, let us also notice that the default DUTM3 model presented in this paper does not verify obtained dynamic changes (whether they are syntactically and semantically correct); but with the proxy, such verification can be done automatically. Finally, all of the operations done by the proxy are completely independent from the target application; *i.e.,* the proxy can download and prepare changes, even when the application is not running. Thus, when the application finally asks the proxy for dynamic changes, the proxy will either immediately supply it with a verified change or do nothing. The last issue concerning the proxy is the way it communicates with a target application. A variety of inter-process communication methods can be used for this purpose, including sockets, message queues, shared memory, and so on. However, if the proxy is a part of the OS, then it can simply write the change to the system memory and then assign that part of the memory (in a form of memory pages) to the target application. We surmise that the OS approach will be the most efficient and natural, though it might be harder to implement.

With the proxy defined, we can now move on to actually implementing the DUTM3 idea in programming languages. First, it is obvious that some changes will be necessary to the language itself; (*e.g.,* a new keyword that will be used to define subtasks). Next, we consider three different types of languages: a) interpreted languages with virtual machines; b) interpreted languages without virtual machines; and c) compiled languages. As a side note: in theory, any language can be implemented in any of these three ways, but usually only one method is chosen (and it affects some properties of the language, like performance or portability).

First, let us consider languages with virtual machines (VMs). In this case, the implementation is the easiest, as VMs are very similar to the DRASP model mentioned above – the intermediary code (bytecode) used by VMs is usually some flavor of assembly language. Moreover, it can include some high-level features not normally used in typical assembler languages. This, in turn, makes it easy to add new instructions that will signify that a new subtask has been reached. The bytecode resides in the memory of VM as regular data, so it is easy to modify without the need to rely on the OS. Of course, the VM needs to comply with DUTM3 rules, applying the change only when new subtasks are reached and after determining whether a change is currently possible or not. Once the code has been altered, the built-in interpreter can execute it as usual. Let us also notice that a Just-In-Time compilation can be used after dynamic changes have been applied (which can increase performance).

In the case of interpreted languages that lack a VM, the situation is a bit more difficult, because the input is a source code that is less convenient than bytecode. Still, the interpreter will either execute source code instruction-by-instruction or transform the source code into some structure like an abstract syntax tree (AST). In the first case, the DUTM3 idea can be implemented by replacing the source code in the memory of the interpreter with the source code obtained from the proxy adviser. In the second case, the proxy needs to supply the interpreter with the dynamic change that is already in the form of the AST, or the interpreter will have to transform the change on its own. Aside from that, the interpreter behaves similarly to the VM, because the program is treated by interpreter as regular data.

The last and most problematic case is compiled languages. Two issues are: a) how to execute the code that will perform the change; and b) how to alter the program that is already compiled and loaded into memory. The first issue can be solved by changing the compiler so that it will include a special code (a function) to every program it generates and will call this code every time a new subtask is reached. While it is possible to add such a function to each program, it would be more efficient to make it a system function that is the part of the runtime library – then, there is only one such function, and it is simply shared by many programs in the form of a dynamic library (`DLL` in Windows or `.so` in Linux, for example).

The second issue is more difficult, as the memory pages marked as executable are not marked as writable (and thus, cannot be easily changed). On the other hand, pages marked as writable cannot be marked as executable, so it is difficult to simply write data to memory and then execute it. However, while regular programs cannot do this, the OS can. We have already used the runtime library to solve the first issue, so we can simply add a function to the OS that will be called as a part of this runtime. Since this is a system function, it can alter the memory freely. Because of this, it is beneficial to implement the proxy as a part of the OS – the communication between the function and the proxy can be simply done in the kernel of the OS. A basic outline of this idea is shown in Figure 8. Of course, this approach will require some changes to both the existing compiles and the OS.

Thus, we have presented the basic methods for implementing the DUTM3 idea in real-life computer systems and programming languages. Let us note that this implementation is possible for any language. However, the implementation would require some changes to existing languages, which may be inconvenient. In such a case, it is possible to create two dialects of the same language (one capable of supporting the dynamic changes, and the other, not).

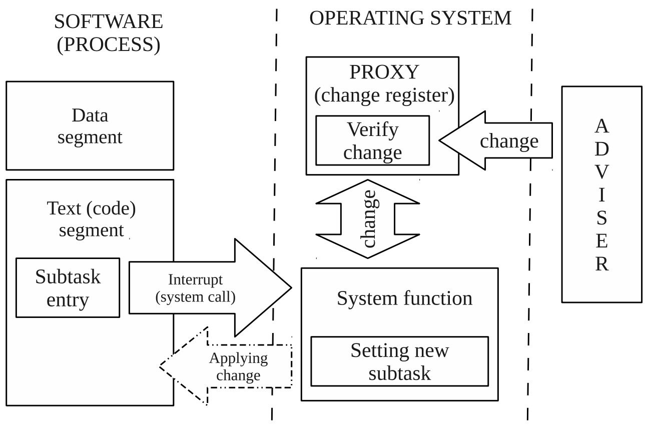

Figure 8. General structure of the dynamic changes system for the compiled languages.

6. Conclusions

We would like to start the conclusions by providing a short summary of the models defined in this paper as well as their relationships with other models existing in the literature. Figure 9 can be consulted for a visual representation of these relationships. For clarity, we will use the full names of the models once again.

We start with the standard Deterministic Turing Machines (DTMs) that can be designed to compute any μ-recursive function (the R-class). The DTM works for any tape alphabet; but for simplicity, we have defined a special case of the DTM that works only for a 3-symbol alphabet. This model was called the on-line 3-symbol Deterministic Turing Machine (DTM3) and was proven to have the same computational power as the original DTM, since any input can be encoded over our 3-symbol alphabet. Thus, in Figure 9, the DTM3 is shown as a special case of DTM.

Regular DTMs or DTM3s can compute only small portion of the R-class, but it is possible to define so-called Universal Turing Machine (UTM) that can simulate any other DTM simply by encoding its program (transition function). The resulting UTM can substitute any other DTM and compute any problem in R-class as long as its supplied a proper program. Moreover, UTM only needs to slower that the DTM it simulates by a constant factor, as the program size is independent from the size of input. For our research we defined the on-line 3-symbol Universal Turing Machine (UTM3) that can simulate any DTM3. In Figure 9 the UTM3 is a special case of the UTM and UTM (UTM3) is a special case of DTM (DTM3) that adds universality to it.

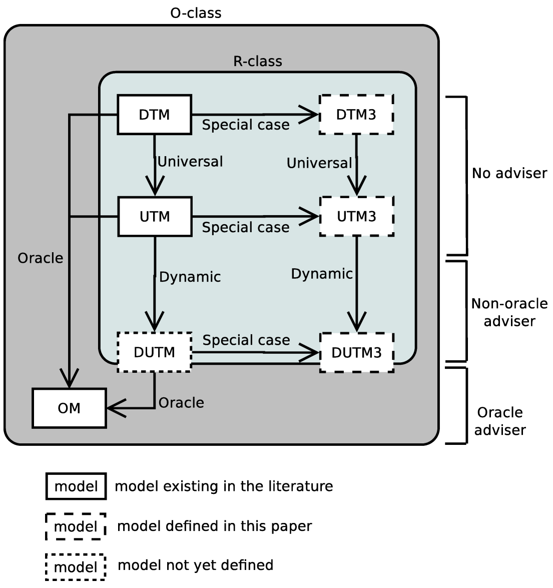

Figure 9. Relations between the Deterministic Turing Machine and similar models.

The main finding of this paper is the definition of the on-line 3-symbol Dynamic Universal Turing Machine (DUTM3) that is connected to an adviser that is capable of supplying the DUTM3 with change descriptions at runtime. We have proven that the DUTM3 needs only $\frac{2}{7}$ more space and $\frac{1}{5}$ more time than the UTM3 to simulate the same DTM3, and that it indeed allows for runtime code changes. In Figure 9, the DUTM3 is a special case of the UTM3 that adds dynamic code properties. We also see that the general version of the DUTM3 that works for any tape alphabet – DUTM – still remains undefined.

We have also discussed that the adviser is similar to Oracle Machine – the Oracle can be understood as a special case of adviser and would allow us to compute problems beyond the R-class – the Oracle-class or O-class. Thus, dynamic models like the DUTM and DUTM3 stand somewhere between the R- and O-classes. However, with Oracle being physically impossible to construct, their practical computational power is restricted to the R-class.

To sum it up, we introduced a dynamic universal model of computation for 3-symbol deterministic Turing Machines (DUTM3) with the capability of applying code changes at runtime in this paper. We defined runtime codes changes and the algorithm for applying them at the level of models of computation; therefore, this concept can be used with any programming language or paradigm based on a model of computation equivalent to a deterministic Turing Machine, including procedural programming (`C`), functional programming (`Haskell`), aspect-oriented programming, object-oriented programming (`C++`, `Java`), prototype-based programming (`JavaScript`), etc.

We have also discussed how the DUTM3 model could be implemented in real-life programming languages.

We confirmed the ability of the DUTM3 to perform runtime changes, and we established and proved a set of properties of the DUTM3 (including its computational power as well as its time and space complexity). In particular, the computational power of the DUTM3 minus the Oracle adviser was proven to be no less than that of the Universal Turing Machine. With the Oracle adviser, on the other hand, the computational power of the DUTM3 matches the computational power of Oracle Machines. In terms of time and space complexity, we proved that, compared to the 3-symbol deterministic Turing Machine it simulates, the DUTM3 model of computation imposes no asymptotic penalties on time and space complexity dependent on the input or its size. However, the model imposes time and space complexity penalties dependent on the number of internal states and subtasks defined in the simulated machine. The upper bound on the computation time of the DUTM3 model of computation compared to the UTM3 – introduced in this paper as well – is 20%, while empirical research suggests a much lower overhead of about 3%. This overhead is also dependent on the density of subtasks in the final program. Moreover, the DUTM3 uses up to 28.6% more space than the UTM3 in the worst-case scenario; in a practical case, this is also dependent on the density of the subtasks.

References

[1] Arora S., Barak B.: *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[2] Chen H., Yu J., Chen R., Zang B., Yew P.C.: POLUS: A POwerful Live Updating System. In: *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pp. 271–281, IEEE Computer Society, Washington, DC, USA, 2007, ISBN 0-7695-2828-7, `http://dx.doi.org/10.1109/ICSE.2007.65`.

[3] Cheng S.W., Garlan D., Schmerl B.R., Sousa J.P., Spitznagel B., Steenkiste P., Hu N.: Software Architecture-Based Adaptation for Pervasive Systems. In: *Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, ARCS '02, pp. 67–82, Springer-Verlag, London, UK, 2002, ISBN 3-540-43409-7, `http://dl.acm.org/citation.cfm?id=648198.751340`.

[4] Cook S.A., Reckhow R.A.: Time bounded random access machines. *Journal of Computer and System Sciences*, vol. 7(4), pp. 354–375, 1973, ISSN 0022-0000, `http://www.sciencedirect.com/science/article/pii/S0022000073800297`.

[5] Davis M.: *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions.* Dover Publication, 1965.

[6] Dmitriev M.: Towards flexible and safe technology for runtime evolution of java language applications. In: *In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001.

[7] Elgot C.C., Robinson A.: Random-Access Stored-Program Machines, an Approach to Programming Languages. *J. ACM*, vol. 11(4), pp. 365–399, 1964, ISSN 0004-5411, `http://doi.acm.org/10.1145/321239.321240`.

[8] Garlan D., Cheng S.W., Huang A.C., Schmerl B., Steenkiste P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, vol. 37(10), pp. 46–54, 2004, ISSN 0018-9162, `http://dx.doi.org/10.1109/MC.2004.175`.

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

[10] Kleene S.C.: Recursive Predicates and Quantifiers. *Transactions of the American Mathematical Society*, vol. 53(1), pp. 41–73, 1943, ISSN 00029947, `http://www.jstor.org/stable/1990131`.

[11] Oreizy P., Medvidovic N., Taylor R.N.: Runtime Software Adaptation: Framework, Approaches, and Styles. In: *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pp. 899–910, ACM, 2008, ISBN 978-1-60558-079-1, `http://doi.acm.org/10.1145/1370175.1370181`.

[12] Parra C., Blanc X., Cleve A., Duchien L.: Unifying Design and Runtime Software Adaptation Using Aspect Models. *Science of Computer Programming*, vol. 76(12), pp. 1247–1260, 2011, ISSN 0167-6423, `http://www.sciencedirect.com/science/article/pii/S0167642310002303`, special Issue on Software Evolution, Adaptability and Variability.

[13] Rudy J.: Performance and Overhead Analysis in Runtime Code Modification. *Journal of Applied Computer Science*, vol. 21(2), pp. 75–89, 2013.

[14] Rudy J.: Turing machine approach to runtime software adaptation. *Computer Science*, vol. 15(3), pp. 293–310, 2014, ISSN 2300-7036.

[15] Rudy J.: Dynamic Random-Access Stored-Program Machine for Runtime Code Modification. *International Journal of Foundations of Computer Science*, vol. 26(4), pp. 441–463, 2015, ISSN 0129-0541.

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

[17] 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, 2001, ISBN 3-540-42264-1, `http://dl.acm.org/citation.cfm?id=646199.681826`.

[18] 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://doi.acm.org/10.1145/1837852.1621621`.

[19] 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, p. 230, IEEE Computer Society, Washington, DC, USA, 2003, ISBN 0-7695-2020-0, `http://dl.acm.org/citation.cfm?id=950785.950888`.

[20] Zhang J., Cheng B.H.C.: Model-based Development of Dynamically Adaptive Software. In: *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 371–380, ACM, New York, USA, 2006, ISBN 1-59593-375-1, `http://doi.acm.org/10.1145/1134285.1134337`.

[21] Zhang J., Cheng B.H.C., Yang Z., McKinley P.K.: *Architecting Dependable Systems III*, chap. Enabling Safe Dynamic Component-based Software Adaptation, pp. 194–211. Springer-Verlag, Berlin, Heidelberg, 2005, ISBN 3-540-28968-2, 978-3-540-28968-5, `http://dl.acm.org/citation.cfm?id=2167692.2167703`.

Affiliations

Jarosław Rudy
Wrocław University of Science and Technology, Department of Control Systems and Mechatronics, ul. Janiszewskiego 11–17, 50-372 Wrocław, `jaroslaw.rudy@pwr.edu.pl`

Received: 4.05.2015
Revised: 27.08.2015
Accepted: 28.08.2015