P ≠ RP Proof

André Luiz Barbosa
http://www.andrebarbosa.eti.br
Non-commercial projects: SimuPLC – PLC Simulator & LCE – Electric Commands Language

Abstract. *This paper demonstrates that P ≠ RP. The way was to generalize the traditional definitions of the classes P and RP, to construct an artificial problem (Maj2/3-XG-SAT) and then to demonstrate that it is in RP but not in P (where the classes P and RP are generalized and called too simply P and RP in this paper, and then it is explained why the traditional classes P and RP should be fixed and replaced by these generalized ones into Theory of Computer Science). The demonstration consists of:*

1. *Definition of Restricted Type X Program*
2. *Definition of the Majority2/3 Extended General Satisfiability – Maj2/3-XG-SAT*
3. *Generalization to classes P and RP*
4. *Demonstration that the Maj2/3-XG-SAT is in RP*
5. *Demonstration that the Maj2/3-XG-SAT is not in P*
6. *Demonstration that the Baker-Gill-Solovay Theorem does not refute the proof*
7. *Demonstration that the Razborov-Rudich Theorem does not refute the proof*
8. *Demonstration that the Aaronson-Wigderson Theorem does not refute the proof*

Mathematics Subject Classification (2010). Primary 68Q15; Secondary 68Q17.

Keywords. P, RP, NP, Computational Complexity, Randomized Algorithms, Derandomization, Formal Languages, Automata Theory.

Contents

1. Introduction

We, following *The Barbosa's Program* and the ideas proposed in [13], could generalize the concepts concerning the class RP (Randomized Polynomial Time) in order to settle the P versus RP question, which is done here. (About this *Program*, see yet [18, 20].)

Accordingly, in Sections 2 and 3 the *restricted type X programs* and the *Maj2/3-XG-SAT* problem are formally defined, and some notes are included to avoid the traps in these definitions. In order to define the *Maj2/3-XG-SAT*, the poly-time computation is redefined in more general form. So, it is proved that the *Maj2/3-XG-SAT* is in RP. Then, in Section 4 it is proved that this problem is not in P (therefore, $\mathbf{P \neq RP}$, naturally, leading to $P \neq BPP$, $P \neq ZPP$, and other great related results), by demonstrating that it is impossible that any poly-time deterministic computation solves the *Maj2/3-XG-SAT*.

In this proof, nothing is assumed about type, structure, form, code, nature, shape or kind of computation, neither structure (or lack thereof) of data, eventually used into any DTM that tries to decide the problem in polynomial time. Otherwise, my proof exploits properties of computation that are specific to real world computers (without *oracles, infinite TMs* and other supernatural devices). In Sections 5, 6 and 7, it is demonstrated that the theoretical barriers against possible attempts to solve the P vs. RP question (since $P \neq RP$ leads to $P \neq NP$, for $P \subseteq RP$ and $RP \subseteq NP$) are not applicable to refute my proof. Finally, in Sections 7 and 8 there are some comments about related work (or lack thereof) to really solve this question, and references, respectively.

Shortly, in order for this $P \neq RP$ proof of mine be accepted, it is sufficient that the fact if there is an L_Z-language (promise problem) separating complexity classes, then they are truly distinct, and the Def. 3.7 are both accepted. On scientific revolution/paradigm shifts, see [18].

2. Definition of Restricted Type X Program

Definition 2.1. Let **S** be a deterministic computer program, let **n** be a finite positive integer and let *time P(n)* be a poly(n) upper bounded number of deterministic computational steps (where time P(n) is not previously fixed for all possible programs **S**, but it is fixed for every one). **S** is a *restricted type X program* if and only if the following three conditions are satisfied:

1. **S** allows as input any **n**-bit word (member of arbitrary length **n** from $\{0, 1\}^+$).

2. The **S** behavior must be for each input one of the following:

   i. **S** returns **0**;
   ii. **S** returns **1**; or
   iii. **S** does not halt (never returns any value).

3. The total **S** behavior must be for each **n** one of the following:

   i. **S** returns in time P(n) **0** for all the 2^n possible inputs of length **n**; or
   ii. **S** returns in time P(n) **1** for at least $\lceil 2^{n+1}/3 \rceil$ possible inputs of length **n**.

Note 1: The presence of **S** is not to be decided – see Section 3.3.1. Testing whether a computer program is a restricted type X program will not be necessary to the proof. **S** will be given as an absolute assumption: It IS a restricted type X program, and this fact will NOT be

under consideration: This is not a contradiction, definitely, since we can easily create innumerous programs of this type and – without need deciding about their types – produce a myriad of instances of the Maj2/3-XG-SAT problem with them – see Sections 4.1 and 4.2, for details.

Note 2: There is no need that the polynomial running times involved in a proof must be previously fixed in order to be defined: For example, what is the fixed polynomial that upper bounds the running time of the reducer concerned in the Cook-Levin Theorem? There is no such fixed polynomial, since this running time depends on the NP problem whose instance is to be reduced to a Boolean formula, but the running time of this reducer is (and must be) polynomial, it is not undefined, of course, otherwise there would be no NP-Completeness. (This insight is formalized in the Def. 3.7.) About this issue, see yet [23] and [24].

Notice that it does not matter at all that we have a different time bound for each NP problem, but the same time bound for each instance of a fixed one, since for this reducer any instance from every NP problem is like just a mere *input* to a deterministic computer program: what is important here, in fact, is that that polynomial time bound is NOT *uniform*, whereas it is – without any contestation – considered very well defined.

Note 3: The running time of a fixed program (or machine) **S** on those inputs for which it halts is bounded by a polynomial P(n) (which is a time-constructible function (for each fixed **S**), evidently [21]), hence there must be an equivalent machine (to each fixed **S**) which always halts, and still runs in deterministic polynomial time, of course. This, however, is not the main point: It is unimportant really whether there must be such an equivalent machine: What matters for my proof, after all, is that this equivalent machine (or program) cannot in general be constructed within deterministic polynomial time, at all, since that polynomial P(n) is *a priori* unknown or not given and cannot be computed within deterministic polynomial time [13].

Note 4: Into the traditional definitions of the classes P and RP, a polynomial P(n) must be fixed for whichever programs **S** (in order to the Maj2/3-XG-SAT problem (Def. 3.1) is in traditional RP), and it is only over the class of all polynomial-time machines that such a polynomial is not fixed. However, into the new definitions of the classes P and RP (Defs. 3.5, 3.6 and 3.7), there is no need that there is a fixed polynomial P(n) for all possible programs **S** in order to the Maj2/3-XG-SAT problem is in the new class RP (Def. 3.5) (see Proposition 3.1). Thus, the comparison with the Cook-Levin Theorem is here quite well placed (in the note 2 above).

3. Definition of Majority2/3 Extended General Satisfiab. – Maj2/3-XG-SAT

Definition 3.1. Let **S** be a restricted type X program and let **n** be a finite positive integer. The problem *Majority2/3 Extended General Satisfiability* (*Maj2/3-XG-SAT*) is the question "Does **S** return value **1** for at least $\lceil 2^{n+1}/3 \rceil$ inputs of length **n**?" Thus, in the Maj2/3-XG-SAT question, the input is the pair $\langle \mathbf{S},\mathbf{1^n} \rangle$, clearly, where $\mathbf{1^n}$ is just **n** in unary form. Note that the specific and fixed time P(n) related to **S** is NOT given at all.

Be careful with a possible confusion made about the Maj2/3-XG-SAT and the Bounded Halting problem (BH), defined over triples $\mathbf{w} = \langle \boldsymbol{M},\mathbf{x},\mathbf{1^k} \rangle$, where $\boldsymbol{M}$ is a nondeterministic machine, **x** is a binary string, **k** is an integer, and $\mathbf{w} \in \mathbf{BH}$ if and only if there exists a computation of $\boldsymbol{M}$ on input **x** that halts within **k** steps [12]: The Maj2/3-XG-SAT is a very different problem, since the time P(n) is not given, and the program **S** into the pair $\langle \mathbf{S},\mathbf{1^n} \rangle$

always halts for at least $\lceil 2^{n+1}/3 \rceil$ inputs of length **n**, but maybe **S** does not halt for all the other ones. Furthermore, the Maj2/3-XG-SAT cannot be reduced in polynomial time to BH (– See the proof of Proposition 4.1). In order to understand why, verify that my Maj2/3-XG-SAT problem is in the new [generalized] class RP (Def. 3.5), by Proposition 3.1, but it is not in that old traditional one.

See, yet, that the Maj2/3-XG-SAT is also so distinct from the PP-Complete problem MajSAT, defined over Boolean formulae. F ∈ MajSAT if and only if more than half of all possible assignments make F true [16]: The Maj2/3-XG-SAT, on the other hand, by Def. 2.1, is an especially diverse problem, since the number of assignments (inputs) that make **S** to return **1** is always either none at all or greater than $2^{n+1}/3$; besides the fact that **S** has a much more complex behavior than a simple Boolean formula and can even not halt for some inputs.

3.1 Definition of well-formed string

Definition 3.2. Let **w** be a string from {**0**, **1**}$^+$. **w** is a *well-formed string* if and only if **w** has the form **1**$^+$**0s** – where **1**$^+$ is a finite positive integer **n** encoded in unary form and **s** is the binary representation of the DTM (deterministic Turing Machine) that simulates a restricted type X program **S**. For **n** = 13, a well-formed string **w** would be, for instance, **1111111111111**0**1001000101001110010010101100100101011001001011110010010110...1**.

3.2 Definition of the Maj2/3-XG-SAT as well-formed string acceptance testing to a language *L*

Definition 3.3. Let ***L*** be a formal language over the alphabet Σ = {**0**, **1**}. A well-formed string **w** ∈ ***L*** if and only if the DTM encoded into **w** returns **1** for at least $\lceil 2^{n+1}/3 \rceil$ inputs of length **n**. The Maj2/3-XG-SAT is the well-formed string acceptance testing to ***L***.

Note that as the size of a restricted type X program **S** is constant on **n** (|**S**|(n) = **c**), the length of the DTM that simulates **S** is constant too on **n** (|**s**|(n) = **k**), and then |**w**| = **n** + **1** + **k**. Thus, time P(n) is the same as time P(|**w**|) and time exp(n) is the same as time exp(|**w**|).

3.3 Class of the language *L* and Class of the L_z-language *L*

L is a nonrecursively enumerable (non-RE or non-Turing-recognizable) language [1], since it is undecidable whether or not an eventual result **1** from a computer program occurs within polynomial time [19], besides the undecidability even whether just it halts for some input [4].

(**Note**: The undecidability of the language ***L*** does NOT contradict the proof. The Maj2/3-XG-SAT is not the undecidable decision problem **w** ∈? ***L***, but just the decidable one **well-formed string w** ∈? ***L***, as explained in Section 3.3.1, since a well-formed string **w** is *given* as an absolute assumption: **w** IS well-formed string, and this fact is NOT under consideration. See that exactly the same kind of statement holds to traditional formal languages, where the absolute assumption is that the strings to be tested are members from Σ* [1].)

Language Incompleteness – The computer theorists generally make a big mistake on definition of *computational decision problem*: They think that ones is the same thing that *languages*, as if all decision problems could be modeled as string acceptance testing to formal languages, like in [1, 5, 6]; however, there exist computational decision problems that can

only be modeled as string acceptance testing to L_z-*languages* (as defined in Section 3.3.1), not to languages, like the Maj2/3-XG-SAT.

Thus, all computer theorists generally say '*problem*' to mean '*language*' and vice versa. See below an excerpt of text of a preeminent Professor in the area, in [10]:

"*By Savage's theorem, any PROBLEM in P has a polynomial size family of circuits. Thus, to show that a PROBLEM is outside of P it would suffice to show that its circuit complexity is superpolynomial.*" [The words *PROBLEM* are lowercased in the original]

However, the set of all languages is a mere proper subset of the stronger and more powerful set of all L_z-*languages* (all the computational decision problems), as established below.

3.3.1 More general definitions for RP and P and definition for L_z-language

Definition 3.4. Let L_z be a language over a finite alphabet, Σ, and let $L \subseteq L_z$. We will call L an L_z-*language*. If $L_z = \Sigma^*$, then L is a Σ^*-language, a *trivial* L_z-*language*, which is the same as language (Σ^*-language = language). The complement of an L_z-language A is another L_z-language $\bar{A} = L_z - A$. Thus, L_z-*language* is simply a generalization to *language* and a string acceptance testing to L is a *computational decision problem* where the string to be tested is *necessarily* member from L_z. If *language* is a *set*, L_z-*language* is a *set into another*.

Observe that a string acceptance testing to L is a *computational decision problem*, but L, rigorously, is not only a *language*, because $L \subseteq L_z$, which is more restrict than simply $L \subseteq \Sigma^*$, which should hold if L was only a language [6]. Thus, all the computational decision problems can be modeled as string acceptance testing to L_z-languages, for to accept a string from any determined subset of Σ^* is much more general than do it just from Σ^*, of course.

The main point here is that the central relevance of the languages is originated in the fact that they model problems, not the inverse. Hence, great part of the Theory of Computation is about languages because of the mistake referred in Section 3.3. When this mistake – that it is said as *mistake* because it leaves legitimate problems out of that old traditional definition – is fixed, the Theory of Computation will certainly study the generalization to language: The richer and stronger concept of L_z-language.

A language over Σ is a subset of Σ^*, and an L_z-language is a subset of the language L_z over Σ. However, as $L \subseteq L_z$ and $L_z \subseteq \Sigma^*$, then $L \subseteq \Sigma^*$, which implies that all L_z-languages are Σ^*-languages, or simply languages, too, naturally. Any language L is also an L-language, and any L_z-language L is also a language L. In fact, if $L_y \supseteq L_z$ then any L_z-language L is an L_y-language L, too. But the great advantage of the L_z-languages is that string acceptance testing to ones can be much easier than to languages, because the strings $\mathbf{x}$ to be tested are in special form: $\mathbf{x} \in L_z$ (this is an absolute assumption). Hence, if we know that all the strings to be tested are from a fixed language L_z, then it is worth to model this problem as an L_z-language; but if we do not know it, we must model it as a simple language, of course.

Consequently, the concept of L_z-language allows the insertion of previous knowledge about the form of the strings to be tested – when they were already constructed in special form or previously accepted by another machine – into traditional concept of language.

(**Note**: If the machine M that decides an L_z-*language* L is fed a string **x** that is in L_z, then M *must* decide whether or not **x** is in L, anyway returning correct answer to **x** $\in$? L; on the other hand, if M is fed any string that is not in L_z, it may do whatever, returning anything, even *incorrect* answer to **x** $\in$? L [Σ^*-*language* L, in this case], or even not halting at all.)

For instance, the language $\{0^n1^n \mid n > 0\}$ over $\{0, 1\}$ is not regular, but verify that if $L_z = \{0^n1^n \mid n > 0\} \cup \{1^n0^n \mid n > 0\}$, for example, then the L_z-language $L_1 = \{0^n1^n \mid n > 0\}$ is regular and can be decided by the NFA $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$, where $\delta(q_0, 0) = \{q_2\}$, $\delta(q_0, 1) = \{q_1\}$, $\delta(q_1, 0) = \emptyset$, $\delta(q_1, 1) = \emptyset$, $\delta(q_2, 0) = \{q_2\}$, $\delta(q_2, 1) = \{q_2\}$, and there are not ε-moves.

Verify that this NFA M recognizes the language $L = 0\{0, 1\}^*$ and $\{0^n1^n \mid n > 0\} = 0\{0, 1\}^* \cap (\{0^n1^n \mid n > 0\} \cup \{1^n0^n \mid n > 0\})$. In fact, this is not coincidence:

Theorem 3.1. If a machine M (DFA, NFA, PDA, DTM, NTM, etc.) recognizes a language L, then M recognizes any L_z-language $L_1 = L \cap L_z$.

Proof. Suppose that a string **x** $\in$ L_z-language L_1 was accepted by a machine M: Then, **x** $\in$ L_z (this is an absolute assumption: All the strings to be tested must be member from L_z) and **x** $\in$ L (the language that M recognizes, regardless of the special form of **x**), which implies that **x** $\in$ $L \cap L_z$; on the other hand, if **x** $\in$ $L \cap L_z$, then **x** will be accepted by M, because **x** $\in$ L_z (**x** can be tested) and **x** $\in$ L (**x** will be accepted by definition of string acceptance testing to languages), which implies that the L_z-language recognized by M $L_1 = L \cap L_z$. □

See, thus, the proposed fix to the traditional formal definition for the class RP (Randomized Polynomial Time) [16]:

Definition 3.5. Let L be an L_z-language recognized by an NTM (nondeterministic TM) M. $L \in$ RP if and only if the following two conditions are satisfied:

1. M accepts **x** $\in$ L_z if and only if the number of the accepting poly-time computation paths on input **x** is at least some nonzero constant fraction **q** (independent of |**x**|) of all the computation paths.
2. When **x** $\in$ L_z and **x** $\notin$ L, all the computation paths of M on input **x** halt within polynomial time into non-accepting state.

Note: When M is formalized as a probabilistic polynomial-time NTM, the probabilities from its behavior are: in (1), Pr[M accepts **x**] = **q** (M halting within polynomial time into accepting state) and Pr[M rejects **x**] = **1 – q** (either M halting into non-accepting state or not halting); and in (2), Pr[M accepts **x**] = **0** and Pr[M rejects **x**] = **1** (M halting within polynomial time into non-accepting state).

See that the class NP, on the other hand, needs only one accepting poly-time computation path (since only one suffices, where **q** in this case is not constant independent of |**x**|, but decreases exponentially on it), which makes the fact that RP is a subset of NP obvious. In addition, the class P needs that the fraction above is 1 (since there is only one path in deterministic computations), which does evident that RP is also a superset of P. Note, yet, that

– as $\mathbf{x} \in L_z$ (this is an absolute assumption, by Def. 3.4) – we do not need to describe what language L_z is allowed here.

Thus, if $\mathbf{x} \in L_z$, $\mathbf{x} \in L$ and we run M on input **x** choosing uniformly at random a computation path for enough large number **r** of times, then M accepts **x** with probability **p** so close to **1** as we want it (with $\mathbf{p} = \mathbf{1 - (1 - q)^r}$, where $\mathbf{r} = \mathbf{\log_{1-q}(1 - p)}$ does not depend on $|\mathbf{x}|$ (since **p** and **q** do not do), which is stronger than the poly($|\mathbf{x}|$) upper-bounded by Chernoff Bound [14], applicable, for example, to class BPP). On the other hand, if $\mathbf{x} \in L_z$, $\mathbf{x} \notin L$ and we run M on input **x**, then M rejects **x** with probability equal to 1, definitely.

Verify that when $L_z = \Sigma^*$ and all the computation paths of M run within old traditional polynomial time, the formal definition above is equivalent to the traditional one for the class **RP** – a set of mere languages –, which implies that this one is just a particular case of the proposed fixed definition. Consequently, we can name the traditional class **RP** as class **Traditional-RP** (or, shortly, **TRP**), where the mathematical truths on the traditional class **RP** continue holding in. Alternatively, we could call the true class **RP** defined above – an actual set of computational decision problems, or L_z-languages – as class **Maj2/3-RP** (or, shortly, **MRP**), for example, but this naming method would be a mistake: A subset would have the name of the set and the set would have a derived name of the subset, which is hard to explain, confuse and damages the clearness of the notation. The same happens with the class **P** in Def. 3.6.

Proposition 3.1. Maj2/3-XG-SAT is in class RP.

Proof. Given **n** and a restricted type X program **S**, the question "Does **S** return a value **1** for at least $\lceil 2^{n+1}/3 \rceil$ inputs of length **n**?" can be decided in nondeterministic polynomial time (time NP(n): as time P(n), using nondeterminism), since can be constructed a universal NTM that simply simulates the running of **S** and tests it for any $\lceil 2^n/3 \rceil$ possible inputs of length **n** at the same time ("on parallel") and verifies in time NP(n) the returns: If they are **0** for all the inputs, then the NTM will answer "No" after the conclusion of the last computation path (branch); on the other hand, if at least one return is **1** then the NTM will answer "Yes" at the end of the first path that returns **1**, regardless of whether the other paths are yet running. One and only one of these two occurrences must happen in time NP(n), by Def. 2.1.

Finally, as the number of the accepting poly-time computation paths of this NTM on input **w** that it is in Maj2/3-XG-SAT is at least 2/3 of all possible ones, and there is no accepting path on input that it is not in it, this L_z-language is in RP, with the constant fraction $\mathbf{q} = 2/3$. □

Note that although the poly-time $T(n) = O(n^{\mathbf{i}})$ that all non-accepting and at least $\lceil 2^{n+1}/3 \rceil$ accepting computation paths spend is a different polynomial for each input **w**, Maj2/3-XG-SAT is a single problem (it is false that any recursive decision problem is poly time reducible to it, since T(n) is not previously fixed for all **S**, but it is fixed for every one, by Def. 2.1 – See the note 1 below, for details), where **i** does not depend on **n**, even though it does on **w**. Consequently, the accepting computation paths is really poly-time one, by Def. 3.7, and the proof above is wholly correct: Maj2/3-XG-SAT is in RP, undoubtedly.

See that Maj2/3-XG-SAT has strings of the form $\mathbf{1^n 0 s}$, where **s** is a DTM simulating a restricted X program **S** that accepts within polynomial time some string of length **n** (returning **1** for at least 2/3 of all **n**-bit inputs). Notice that we do NOT need to check whether **S** is a restricted X program, by Def. 3.4.

(**Note 1**: Suppose that someone says that the Maj2/3-XG-SAT is not in RP, since its complexity class is really undefined, and it can be, for example, EEXP-Hard (for double exponential time), reasoning as below:

"Let L be an EEXP problem, M be the deterministic Turing Machine that solves L in time $t(n) = 2^{2^{\{poly(n)\}}}$. Then we can reduce L to Maj2/3-XG-SAT as follows: Given an input **x** for the problem L, we construct a program **S** that ignores its input and simulates M on input **x**. The promise is satisfied by the constant polynomial p(n') = t(|x|), and clearly (**S**,1) is an instance of Maj2/3-XG-SAT if and only if M accepts **x**."

Fortunately, constructions like above cannot disprove that Maj2/3-XG-SAT is in RP, since they do not take into account that time P(n) is not previously fixed for all possible programs **S**, but it is fixed for every one, as stated in Def. 2.1 – hence, as $2^{2^{\{poly(|x|)\}}}$ is not upper bounded by any fixed poly(n), that program **S** is not a restricted type X program, and clearly (**S**,1) is NOT an instance of the Maj2/3-XG-SAT.

Finally, see also that the function $2^{2^{\{poly(|x|)\}}} = t(|x|)$ is not constant, but depends on |x|. However, if **x** is fixed into that TM M simulated by **S**, then this function is a constant (and then M halts on **x** within only $O(1)$ steps, since M and **x** are fixed independent of **n**); nonetheless, in this case, M does not solve L, of course, and then the disproof above fails.)

(**Note 2**: Suppose, yet, that anyone else says that the Maj2/3-XG-SAT is not in RP, since Proposition 3.1 is wrong, as long as either no poly-time TM can simulate a universal TM, or it – about the universal NTM that simulates the running of **S** – does not consider the running time of this simulation, which could be nondeterministic non-polynomial.

Fortunately yet, these refutations of Proposition 3.1 are equivocated, since a program **S** is always restricted (hence, it is NOT a universal TM), and the nondeterministic running time of the simulation of the program **S** (encoded into **x**) running for any $\lceil 2^n/3 \rceil$ possible inputs of length **n** at the same time IS necessarily (must be) nondeterministic polynomial, since *time P(n)* is a time-constructible function (for each fixed **S**), by Def. 2.1.

See, however, this interesting review:

"– The author proposes that Maj2/3-XG-SAT is in (promise-)NP but not in (promise-)P. He is right about the second part, but incorrect about the first part: Maj2/3-XG-SAT is unconditionally not in promise-NP. He gives a simple but fallacious argument that Maj2/3-XG-SAT is in promise-NP on p. 8. In note 2 on p. 8 he anticipates but rejects a counterargument, but he is wrong and this counterargument is essentially correct.

The reason is as follows: for any Turing machine *M* and positive integer *t*, we can form a machine M_t that outputs **0** on all inputs except those of length *t*, on which it behaves like *M*. If *M* always halts and *M*'s behavior depends solely on its input length (call this latter restriction *semi-blindness*), then M_t is always a restricted type-X machine.

It is known there exists a unary language *L* that is decidable, yet it is not in EXPTIME, hence not in NP. There is a *semi-blind* machine *M* that decides *L* correctly on each input having the form *1^t*. But if Maj2/3-XG-SAT were in promise-NP, then we could solve *L* in NP: given input *x* of form *x = 1^t*, we decide whether *x* is in *L* by running the presumed NP verifier on the input (M_t, *1^t*), which obeys the promise. (If *x* is not of form *1^t*, then we can reject *x*.)"

Verify that that conclusion is not true: In order to try to decide that language L in NP, as proposed above, we must run the NP verifier on the inputs of form (M, *1^t*), not (M_t, *1^t*), since to solve whether M accepts *1^t* is quite different from do it about M_t, for M is not the same thing neither has the same running time complexity as all the machines M_1, M_2, M_3, ... taken into account as a [countably infinite] set. The running time of all those M_t is only $O(1)$, since t is a fixed constant into M_t, independent of **n** (|input|), while *L(M)* is not even in EXPTIME (hence, M is not a restricted type-X machine, at all), which implies, fortunately, that the input (M, *1^t*) does not obey the promise in Def. 2.1, and then L cannot be decided in NP as proposed by that smart reviewer, and then the disproof above fails too.

See, also, another interesting and similar review:

"– Let L be any computable language, encoded in unary, and M a deterministic TM that solves L. The program S = S_x takes its input **y**, and compare its length to **x**. If |y| = |x|, then S(y) simulates M on input **x**, and, if M(x) accepts, S(y) accepts. Otherwise, S(y) rejects. If |y| is any other value, S(y) rejects.

Clearly, this S runs in linear time, since all it has to do is count the length of y, **except** when |y| = |x|, but this is only finitely many exceptions, and hence doesn't change the asymptotic running time of S. To reduce L to Maj2/3-XG-SAT: map **x** to the pair (S_x, 1^{|x|})."

Verify that that conclusion is not true too: By means of the same reasoning above, we could prove that that language L would be in NP, since S_x and its input may be mapped to a Boolean expression in deterministic polynomial time (for the running time of S_x is really only a fixed constant), and then this contradiction shows that this other disproof fails too.)

In fact, all complexity classes can be generalized with the concept of *L_z-language*, like this new definition proposed for the class **P**:

Definition 3.6. Let L be an L_z-language. $L \in \mathbf{P}$ if and only if for all x $\in L_z$, x $\in$? L is decidable by a poly-time DTM. Be careful with the traps: For example, all ***L_z*-languages** L_z are trivially in **P** (where L_z can be *any* language, even non-Turing-recognizable ones), which does NOT mean that all **languages** L_z (Σ^*-languages) are in **P**, noticeably.

Notice that the proper definition of *polynomial-time computation* is generalized here, without losing its more important characteristic: To be understood loosely as "feasible in practice", where the critique in [25] is not applicable:

Definition 3.7: Poly-time computation. A computation is said to be polynomial-time if its running time T(n) = $O(n^k)$, where k = $O(1)$, even that **k** depends some way on input. (n = |input|.)

Into the old traditional definition, **k** must be a fixed constant (that does not depend on **n**, obviously), but this stronger restriction is not essential to the vital matter: To maintain the character of vaguely practicable for deterministic polynomial-time computations. In Maj2/3-XG-SAT, **k** depends on the **S** encoded into **w**, but it is in $O(1)$, since that even that **k** cannot be computed [1] neither is given, it is a fixed constant for each fixed **S**, by Def. 2.1. Furthermore, the traditional definition of polynomial time computations asserts a hidden assumption: **k** must be *a priori* a <u>*known*</u> and <u>*given*</u> fixed constant, as revealed in Section 3.4.1.1 in [13].

See that if T(n) = $O(2^{poly(n)})$, for example, then T(n) = $O(n^k)$, where **k** (poly(n) $\log_n$ 2) is not in $O(1)$, evidently, and is upper unbounded (for non-constant poly(n), of course): hence, in this case T(n) is not polynomial at all. The same happens with T(n) = $O(n^{\log n})$. If T(n) = $O(n^k)$, where **k** is, for example, the [arbitrary] position of the first 1 in **w** (or 1, if **w** = 0^n), then **k** is not in $O(1)$ too, for those possible positions can be from 1 to |**w**| = **n**, hence in the extreme case T(n) = $O(n^n)$. On the other hand, if T(n) = $O(n^{g(n)})$, but now g(n) is upper bounded by a finite positive constant **k**, that is $\lim_{n\to\infty} g(n) < k$, then T(n) = $O(n^k)$, whence it is polynomial.

Some experts are asserting: "– The Maj2/3-XG-SAT is not in RP (in the author's terms): the polynomial n^k CANNOT depend on the input." However, this assertion is false, being true only for the old traditional definition of poly-time computation, since that in the new definition (Def. 3.7), the polynomial CAN definitely depend on the input – as long as that **k** is in $O(1)$. Think: This is just a matter of Math object definition, not of mathematical error or correctness, at all. We are not obligated to follow obsolete definitions only because they are established, unless the Science is finished (or dead). See Section 8.

Very important: Verify that these new definitions of the classes P and RP are simply good generalizations of the old traditional ones: Any traditional P or RP problem IS too, respectively, in the new class P or RP defined above (even though the converse is in general false, since these new generalized classes are strictly larger than the traditional ones), and any superpolynomial deterministic or randomized problem is NOT in the new class P or RP, respectively, which proves that these generalizations are consistent and smooth.

3.3.2 L_z-languages and Promise Problems

An *L_z-language* L can be considered as a *promise problem* $\prod$, as introduced by Alan L. Selman [Information and Computation, Vol. 78, Issue 2, (1988), pp. 87-98] and defined in [9], where the *promise* ($\prod_{YES} \cup \prod_{NO}$) = L_z, $\prod_{YES}$ = L, $\prod_{NO}$ = $L_z - L$, and its restricted alphabet {0, 1} is generalized to any finite alphabet Σ. Nonetheless, notice that the concepts, notation, generality, power and applicability of the *L_z-languages* are clearer, richer, simpler, conciser, more elegant, aesthetic and stronger than ones of the *promise problems*.

4. Demonstration that the Maj2/3-XG-SAT is not in P

Theorem 4.1. P ≠ RP.

Proof. As demonstrated in Sections 3.3.1, any instance of the Maj2/3-XG-SAT can be recognized in nondeterministic polynomial time and in randomized polynomial time. However, can it be recognized in deterministic polynomial time?

By hypothesis, consider that it can: In this case, must exist a DTM **Q** that – given a positive integer **n** and a restricted type X program **S** into **w** – answers correctly within polynomial time the question "Does **S** return a value **1** for at least $\lceil 2^{n+1}/3 \rceil$ inputs of length **n**?" (If **w** is in Maj2/3-XG-SAT, then **Q**(**w**) = "Yes", else "No"). **Note**: all the inputs for the program **S** in this Section are of length **n**.

Proposition 4.1. The DTM **Q** is, in fact, a real computer program. Although it may work entirely in a different way from someone would expect from the method that the Maj2/3-XG-SAT was defined, **Q** cannot be a magical or dream machine, since it must be an actual machine.

So, let $\mathbf{W}$: $\Sigma^* \times L_z \rightarrow \mathbf{N}$ be a function with a DTM and a well-formed input for it as arguments, where if $\mathbf{W}(\mathbf{Q}, \mathbf{w}) = \mathbf{m}$ (**Q** can simulate the running of **S** into **w** and test some inputs for **S** in such a simulation – considered here a step-by-step process running **S** into **Q**), then **m** is the number of inputs for **S** simulated by **Q** in this process: $0 \leq \mathbf{m} \leq 2^n$.

Note: It does not matter for this proof whether **W** is a computable function or not; and if **X** is not a DTM or is not interested in the Maj2/3-XG-SAT problem, then $\mathbf{W}(\mathbf{X}, \mathbf{w})$ is defined as 0.

Thus, in order to answer the question, there are no miracles: **Q** can act into only four possible ways (where $\mathbf{m} = \mathbf{W}(\mathbf{Q}, \mathbf{w})$):

1. **Q** simulates the running of **S** for:

   i. $\lceil 2^n/3 \rceil$ inputs or more ($\lceil 2^n/3 \rceil \leq \mathbf{m} \leq 2^n$);
   ii. All the inputs from an arbitrary nonempty proper subset of all them with less than $\lceil 2^n/3 \rceil$ inputs ($0 < \mathbf{m} < \lceil 2^n/3 \rceil$); or
   iii. Only one input (or all from a nonempty proper subset of all them with less than $\lceil 2^n/3 \rceil$ inputs) previously computed whose return decides the question ($\mathbf{m} = \mathbf{d} < \lceil 2^n/3 \rceil$).

2. **Q** does not simulate the running of **S** at all ($\mathbf{m} = 0$).

Proof. These ways are exhaustive: Either **Q** simulates the running of **S** or not; and, if **Q** simulates the running of **S**, then it can test on it $\lceil 2^n/3 \rceil$ inputs or more (1.i); arbitrarily less than these ones (1.ii); or just one (or all from a nonempty proper subset of all them with less than $\lceil 2^n/3 \rceil$ inputs) that was anyway previously computed whose return decides the question (1.iii). Unfortunately, there are no more alternatives besides that ones. (Note: Into ways (1.ii) and (1.iii), **m** must be polynomial in **n** in order to **Q** can decide the Maj2/3-XG-SAT in deterministic polynomial time, of course.)

As well, the running time of a universal NTM that decides in time NP(n) the Maj2/3-XG-SAT – as in the proof of Proposition 3.1 – cannot be upper bounded by any fixed poly(n). Moreover, a program **S** does not necessarily halt for all its possible inputs. Furthermore, the time P(n) in Def. 2.1 cannot be upper bounded by any fixed poly(n), too. Thus, in general, cannot exist any fixed poly(n) number of TM configurations that represents the entire processing of **S**.

Additionally, as to find the input whose return decides the question and simulate the running of **S** only for this input is impossible (see in Way 1.iii below), the particular fixed running time P(n) of a specific restricted type X program cannot be computed in any fixed poly(n) upper-bounded number of deterministic computational steps.

Hence, an instance of the Maj2/3-XG-SAT cannot be reduced within polynomial time into another one of a P(n)-time decidable problem, because the reducer machine must run within polynomial time in this case, but it cannot previously know or compute what upper bounds that time P(n), by Proposition 3.2 in [13]. □

Suppose that someone claims, with the following argument, that the P ≠ RP proof of mine fails:

"– The author assumes that the 4 ways mentioned are the only way to solve the problem. Why can't the DTM **Q** decide the question some other way?"

The answer is not complicated: **Q** cannot decide the question by some other way because there is no another possible way to decide the Maj2/3-XG-SAT besides the four ones mentioned above: These ways do not specify type, structure, form, code, nature, shape or kind of computation, neither structure (or lack thereof) of data – but just the **number (m) and kind of inputs (arbitrary or computed) tested** in eventual simulated running of **S** –, into *any* running of *any* DTM that tries to decide the Maj2/3-XG-SAT: (1) $\lceil 2^n/3 \rceil$ inputs or more ($\lceil 2^n/3 \rceil \leq \mathbf{m} \leq 2^n$); (2) arbitrary ones less than $\lceil 2^n/3 \rceil$ inputs ($0 < \mathbf{m} < \lceil 2^n/3 \rceil$); (3) computed ones less than $\lceil 2^n/3 \rceil$ inputs ($\mathbf{m} = \mathbf{d} < \lceil 2^n/3 \rceil$); or (4) none (there is no simulating **S** at all) ($\mathbf{m} = 0$).

Can there be some other way? No, by a reasoning similar to *pigeonholes* from *pigeonhole principle*: Either **Q** simulates **S** or not. And simulating **S** for more than all inputs – or for any subset with exponential number of them – leads to exp(n) running time, as explained in the Way 1.i; less than none go to negative number of inputs, which makes no sense in actual computations; and between these limits the number and kind of inputs for eventual simulated running of **S** must be one from the four mentioned above. Consequently, all the possible deterministic computations to decide the Maj2/3-XG-SAT are really into one from these four ways.

Can we create new ways to decide in deterministic polynomial time the Maj2/3-XG-SAT combining the four ones? Unfortunately, no way: The way 1.i is useless to decide in deterministic polynomial time the question; the way 1.ii is useless to decide in any time the Maj2/3-XG-SAT; and the combination of the ways 1.iii and 2 results simply in the way 1.iii – when none result from the simulation is used by **Q** in order to answer the question, which is a case treated below in the way 1.iii.

Hence, claims like above do not go to refute this P $\neq$ RP proof.

Note, yet, that the method utilized in this proof cannot be adapted to decide whether SAT is in P, because if a program **S** is simulating a Boolean formula with **n** variables, it *must* always halt for all the possible inputs, and its running time *must* be the same for any input; however, these additional restricted conditions cannot be held in general restricted type X programs, like ones in the proofs of the Props. 4.2 and 4.3 below.

Hence, to decide whether an arbitrary general deterministic computer program computes determined output for determined input (which is undecidable, by the Rice's Theorem [11]) cannot be reduced to SAT as it does to Maj2/3-XG-SAT (as demonstrated in these proofs), and then any attempt to adapt my proof to solve whether SAT is in P is condemned to fault.

See that if **S** is simulating a Boolean formula with **n** variables, then the Rice's Theorem cannot be applied to the **S** behavior for any input, since it is restricted for all the possible ones.

Finally, suppose that else one tries to refute the proof saying:

"This proof follows a common theme: Defines an RP problem with a certain structure, argues that any algorithm that solves that problem must work in a certain way and any algorithm that works that way must examine an exponential number of possibilities. But we can't assume anything about how an algorithm works. Algorithms can ignore the underlying semantic meaning of the input and focus on the syntactic part, the bits themselves."

As in the previous "refutation" of my proof, the answer is also not complicated: If the DTM **Q** ignores the underlying semantic meaning of **w** and focus on its syntactic part, the bits

themselves, considering **w** just a series of bits, this approach only places **Q** into the Way 2, where the proof continues to hold, naturally.

Shortly, the spirit of the proof is very simple: the Maj2/3-XG-SAT is decidable by brute-force search because whether **S** returns **1** for at least 2/3 of all the 2^n possible ones is decidable, whereas whether **S** returns **1** for at least one from a subset with less than this quantity (2/3 of $2^n = 2^{n+1}/3$) is in general undecidable (since **S** can even not halt for any input from such a subset), by Def. 2.1, which does that all the other ways to decide the Maj2/3-XG-SAT (without brute-force neither randomized searching) be absolutely hopeless.

Consequently, we can say that one of the profoundest questions in Complexity Theory was solved by this plain ingenious characterization, the Def. 2.1!

Be brave and see below that all these four exhaustive ways to decide in deterministic polynomial time the Maj2/3-XG-SAT fail:

Way 1.i **Q** simulates the running of **S** for $\lceil 2^n/3 \rceil$ inputs or more ($\lceil 2^n/3 \rceil \leq \mathbf{m} \leq 2^n$):

The obvious way to implement the DTM **Q** is to construct a universal DTM that simulates the running of **S** and submits to it any $\lceil 2^n/3 \rceil$ or more inputs, verifying whether it returns **1** for at least one (in a breadth-first search, to avoid running forever in a computation path that does not halt): If all returns are **0**, then **w** is not in Maj2/3-XG-SAT; otherwise, then it is.

Nevertheless, this brute-force method, on worst case, can decide the problem only at the end of testing at least $\lceil 2^n/3 \rceil$ inputs, in time exp(n).

Even though the probability of answer incorrectly after the test of only 32 inputs (chosen uniformly at random), for example, is very very low: Less than $\mathbf{3}^{-32}$ (independent of input size), when **S** does not return **1** for any of these 32 inputs, and then **Q** does not answer "Yes", but **w**, very very unfortunately, is really in Maj2/3-XG-SAT.

Way 1.ii **Q** simulates the running of **S** for all the inputs from an arbitrary nonempty proper subset of all them with less than $\lceil 2^n/3 \rceil$ inputs ($0 < \mathbf{m} < \lceil 2^n/3 \rceil$):

Note that to simulate the running of **S** only for a polynomial number of arbitrary inputs (or just for a number of them less than $\lceil 2^n/3 \rceil$ ones – for instance: $\mathbf{n}^{\log n}$) does not work: Even the test of $\lceil 2^n/3 \rceil - 2$ inputs on the simulation cannot help to decide whether **S** returns **1** for some from the not simulated ones (in fact, this simulation cannot help to decide even whether **S** simply halts for a specified input from these ones).

Moreover, even the simple question whether **S** halts for at least one input from an arbitrary nonempty proper subset of the set of all the 2^n possible inputs with less than $\lceil 2^n/3 \rceil$ ones is undecidable, of course, by Def. 2.1. (Obs.: This question is only decidable for a subset with at least $\lceil 2^n/3 \rceil$ inputs: The answer is always "True", by Def. 2.1.)

Way 1.iii **Q** simulates the running of **S** only for an input (or inputs) previously computed ($\mathbf{m} = \mathbf{d} < \lceil 2^n/3 \rceil$):

Proposition 4.2. A DTM **Q** cannot compute, without simulating the running of **S** for at least $\lceil 2^n/3 \rceil$ inputs, a nonempty proper subset of ones, where the return from **S** for one of them decides the question, and then to simulate the running of **S** only for these inputs to decide the Maj2/3-XG-SAT.

Proof. Let a well-formed string **f** be constructed with an arbitrary **n,** and let the restricted type X program **F** be below, where **Q** was, by the Turing-Church Thesis, translated into a computer program where the instructions Simulated_by_Q[e] := True; and Number_of_Simulated_Inputs := Number_of_Simulated_Inputs + 1; were included just before any instruction of this program that starts the simulation of **F** for any input **e** (Simulated_by_Q and Number_of_Simulated_Inputs are global variables of type dynamic array or vector of Booleans values and integer that were initialized with False in all its positions and 0 (zero), respectively).

We call **Q'** to this program derived from **Q**. Verify that if **Q** runs in polynomial time, then **Q'** also do it, of course, and the behaviors and results from **Q'** and **Q** are the same.

```
01. F(string input) // F is a restricted type X program, since Q' and R are supposed poly-time
                    // DTMs, and F will either return only 0's or at least ⌈2^(n+1)/3⌉ 1's
02. { n := length(input);
03.   if (R(input) = "Yes") return(0); // R returns always within polynomial time "Maybe"
04.   if (R(input) = "No")  return(1); // Thus, R does not matter to the behavior of F
                                       // But Q does not know it: See its work is very hard!
05.   concurrent_ifs // only one of the returns can close the concurrent instructions block
06.   {              // below, where the two if's run concurrently
07.     { if (Simulated_by_Q[input])  return(0); } // There will be d of these returns …
08.     { if (Number_of_Simulated_Inputs > 2^n/3)  return(0); } … and 2^n-d of …
09.     { if (Q'(f) = "Yes")          return(0); else return(1); } … these ones
10.   }
11. }
```

Now, if **Q** does not simulate the running of **F** for at least $\lceil 2^n/3 \rceil$ inputs, then it will unavoidable answer incorrectly "No", after **F** returns **0** for all the simulated inputs, since **F** will in this case return **1** for all the non-simulated inputs (there is at least $\lceil 2^{n+1}/3 \rceil$ ones, by Proposition 4.2), because for these ones there will be chance for the third **concurrent_if** (line 09) to detect at some moment the answer "No" from **Q'**, and then to return **1** (note that, as the Number_of_Simulated_Inputs < 2^n/3, the second **concurrent_if** (line 08) is irrelevant in this case). Note that **Q'** cannot answer "Yes", because **F** will always return **0** for all the **d** simulated inputs, by the first **concurrent_if** (line 07), and the answer from **Q'** is based in the returns from **F** for all the simulated inputs. See that the contents of the global variables Simulated_by_Q (line 07) and Number_of_Simulated_Inputs (line 08) are known into **F**, because **Q'** is concurrently running (line 09).

On the other hand, if **Q** simulates the running of **F** for at least $\lceil 2^n/3 \rceil$ inputs, then it answers correctly "No", after **F** returns **0** for all simulated inputs, since **F**, in this case, returns **0** for all ones because the return of all **0** is by the first and second **concurrent_if** (since now the Number_of_Simulated_Inputs > 2^n/3), when then there is no chance for the third **one** to detect the answer "No" from **Q'**, and then to return **1**. Unfortunately, **Q** can, by this way, decide the Maj2/3-XG-SAT just in time exp(n), as was treated in Way 1.i.

See that if **Q** does not simulate the running of **S** for any input at all – or if none result from the simulation is used by **Q** in order to answer the question –, then it will inevitably

answer incorrectly at some moment, by diagonalization that exists into string **f**, on the third **concurrent_if** (line 09) of **F**.

Finally, suppose that **Q** could decide whether there is diagonalization into string **f**. In this case, **Q** could stay running forever, without simulating the running of **F** (or simulating it and no using the results **0** from this simulation to answer "No") and no returning anything at all, which would imply that **f** is not a well-formed string, and then **Q** would not be incorrect. Alternatively in this case, **Q** could attempt to decide the question either without simulating the running of **S** for any input at all, or simulating it for some inputs and ignoring the results **0** from this simulation (there is no result **1**, of course), either considering thereby **w** just a bit string, or engaging in more indirect reasoning about the code of **S**, as in the Way 2 below.

However, **Q'** can in general be any arbitrary deterministic program and can compute using or not the value of ***input*** for **F**, besides its proper input (the string **f**). Here, if **Q'**(**f**) runs in polynomial time either returning always "Yes" for all values of ***input*** not simulated by **Q**, or returning another result for at least $\lceil 2^{n+1}/3 \rceil$ inputs, then **f** is a well-formed string (independently of the behavior of **Q**(**f**)), and there is no diagonalization into it. Consequently, there is diagonalization into string **f** if and only if **Q**(**f**) = **Q'**(**f**) independently of the value of ***input***.

Hence, if **Q** can decide whether there is diagonalization into string **f**, without simulating the running of **S** for at least $\lceil 2^n/3 \rceil$ possible inputs, then **Q** can decide whether the language of a given arbitrary TM (**Q'**) has a particular nontrivial property (**Q'** accepts **f** if and only if itself (**Q**) does it; in other words, **Q**(**f**) = **Q'**(**f**) independently of the value of ***input***). However, this problem is undecidable, by the Rice's Theorem (reflect: **Q'** could be *any* computer program). Hence, **Q** cannot decide it; thus, **Q** is condemned to fault, too: without knowing neither computing whether there is diagonalization into string **f**, to answer incorrectly the question, by the diagonalization above; or to simulate the running of **F** for at least $\lceil 2^n/3 \rceil$ inputs, in time exp(n). □

Note that, in general, the Rice's Theorem can be applied to the **S** behavior for a nonempty proper subset with at most $2^n/3$ inputs (because this behavior is arbitrary, by Def. 2.1: in this case, **S** can not halt for any input from this subset), but cannot do it to the total **S** behavior for a subset with more than $2^n/3$ inputs (since this behavior is restricted, by Def. 2.1: in this case, either all results from **S** are **0** or at least one is **1**).

Way 2. <u>**Q** does not simulate the running of **S** at all (**m** = 0):</u>

If the running time of **Q** depends on the one of the restricted type X program **S** into **w** for some input (where if **S** does not halt for any input, then **Q** does not halt at all, too), which occurs when **Q** acts reducing **w** into instance of another problem or simulating the running of **S**, then the use of the diagonalization method in order to demonstrate that **Q** cannot decide the problem fails, since **Q** does not have to be as restricted as **S**. But, as these running times are independent ones in the special case treated here, where **w** is considered either just a bit string, or **Q** decides whether **S** returns **1** for some input by engaging in more indirect reasoning about the code of **S**, without simulating it at all, then we can use diagonalization in order to demonstrate that **Q** cannot decide the Maj2/3-XG-SAT. E.g., if **Q** converts a problem in **E** without $2^{n/10}$-size circuits into a PRG which fools n^c-size ones, for any fixed **c**, then **Q** is here.

Note that if the running time of **Q** is anyway always greater than one of the restricted type X program **S** into **w** for some input (where, remember, if **S** does not halt for any input, then **Q** does not halt at all, too), then **Q** is, maybe indirectly, simulating the running of **S** for

this same input or reducing the instance of the Maj2/3-XG-SAT constructed with **S** to some instance of another problem, of course. In general, to reduce within polynomial time an instance of the Maj2/3-XG-SAT is impossible, by the proof of the Proposition 4.1. We will see below that to decide the Maj2/3-XG-SAT without simulating the running of **S** – or do it in a running time upper bounded by any fixed (or even non-fixed) integer polynomial function of |**w**| – is impossible, too:

Proposition 4.3. A DTM **Q** cannot, without simulating the running of **S** for any input, decide the Maj2/3-XG-SAT problem.

Proof. Let **S** encoded into **w** be the program:

```
01. S(string input)
02. {
03.   n := length(input);
04.   if (integer(input) > 2^n/3) { if (T(w) = “Yes”) return(0); else return(1); } else return(0);
05. }
```

Where **T** is an arbitrary deterministic program. Hence, if **Q** can, without reducing or simulating the running of **S** for any input, decide the Maj2/3-XG-SAT problem, then it can decide whether the language of **T** has a particular nontrivial property: **T**(**w**) != **Q**(**w**), since otherwise then **Q** cannot answer anything within polynomial time, because whatever it answers will be incorrect, by diagonalization that holds in this case, in line 04 (**S** returns **0** for all inputs less than 2^n/3, and the answer from **T**(**w**) is inverted and returned by **S** when it process an input greater than 2^n/3: See that there are $\lceil 2^{n+1}/3 \rceil$ inputs of this form. Thus, if **T**(**w**) returns ***“Yes”***, then **w** is not in Maj2/3-XG-SAT, and vice versa). However, if **T**(**w**) != **Q**(**w**) and **T**(**w**) runs in polynomial time, then there is no diagonalization into **S**, and **Q** must decide what **T**(**w**) returns to decide the question. That is, before **Q** answers whatever within polynomial time, it must decide whether **T**(**w**) = **Q**(**w**), to avoid that the diagonalization above forces it to error (reflect: **T** can be *any* computer program).

Nevertheless, this problem is undecidable, by the Rice's Theorem; hence, the DTM **Q** cannot, without reducing or simulating the running of **S** for any input, decide the Maj2/3-XG-SAT. □

Observe that if **Q** tries to test whether the string **w** above is in Maj2/3-XG-SAT (where **Q**(**w**) can return: “Yes”; “No”; or it does not halt) simulating the running of **S** for all the possible inputs, then **S** returns in time P(n) **0** for all ones less than 2^n/3, but, if **T**(**w**) = **Q**(**w**), then the simulation of the running of **S** never halts and never returns any value for any input greater than 2^n/3, by infinite regress, which implies that **Q** does not halt and never answer incorrectly, even though without deciding whether **T**(**w**) = **Q**(**w**), obviously, since it remains forever waiting the return from this simulation for any input greater than 2^n/3, deluded without knowing that it never does. Notice that if **T** answers anything within polynomial time, then **w** is a well-formed string; otherwise, then it is not.

Perceive that to state that **Q** answers whatsoever if and only if **w** is a well-formed string does not work, because, as demonstrated in Section 3.3, ***L*** (the set of all well-formed strings) is a non-RE language, which implies that **Q** cannot decide whether **w** is a well-formed string in order to decide accordingly whether it can answer anything without mistaking. That is, in order to **Q** works in this case, it must assume absolutely that **w** is a well-formed string, and then this assumption implies that it is really true, and that **Q** for the input **w** returns within polynomial time incorrect answer, by the diagonalization above into **S** (line 04).

Proposition 4.4. A DTM **Q** with running time upper bounded by $f(|\mathbf{w}|)$, where f is a fixed [or even unfixed] integer polynomial function of |**w**|, cannot decide the Maj2/3-XG-SAT.

Proof. Let **S** encoded into **w** be the program:

```
01. S(string input)
02. {
03.   n := length(input);
04.   if (integer(input) > 2^n/3) {
05.     concurrent_ifs {
06.       if (T(w) = "Yes") return(0); else return(1);
07.       if (Timer > |w|^k)   return(1);
08.     }
09.   }
10.   else return(0);
11. }
```

Where **T** is an arbitrary deterministic program, **k** is an arbitrary fixed finite positive integer, ***Timer*** counts (in another "thread" or "concurrent process") the running time of **S**, and the two internal ***if*** are evaluated concurrently while **T** runs for the input **w**. Thus, as the functions ***length*** and ***integer*** are poly(n)-time, **S** is a restricted type X program, regardless of the behavior of **T**. See that, as **S** returns **0** for all inputs less than 2^n/3, if **T**(**w**) answers in running time less than $|\mathbf{w}|^{\mathbf{k}}$, then its answer is inverted when **S** process any input greater than 2^n/3, and **T** is forced to the error if it tries to be a decider for Maj2/3-XG-SAT.

Therefore, as **T**(**w**) can be equal to **Q**(**w**) – (if **Q** and **T** are equivalent, for instance) – even though **Q** cannot decide whether it is true that **T**(**w**) = **Q**(**w**), by the Rice's Theorem (reflect: **T** can be *any* computer program) –, this implies that, for large enough **k** (when $|\mathbf{w}|^{\mathbf{k}} > f(|\mathbf{w}|)$, **Q** fails: It cannot know or compute that it cannot in this case answer correctly the question in running time less than $|\mathbf{w}|^{\mathbf{k}}$, by the diagonalization above. Hence, **Q** is again condemned to fault: To answer incorrectly the question before $|\mathbf{w}|^{\mathbf{k}}$ computational steps, for some large enough **k**. Note yet that **Q** cannot adjust $f(|\mathbf{w}|)$ in order to it is always greater than $|\mathbf{w}|^{\mathbf{k}}$, since this polynomial is *a priori* unknown or not given and – by Proposition 3.2 – it cannot be computed within deterministic polynomial time. □

Observe again that if the DTM **Q** decides the Maj2/3-XG-SAT simulating the running of **S**, then **Q** cannot run in time upper bounded by any fixed polynomial function of |**w**| (in fact, none TM – DTM or NTM – that decides the Maj2/3-XG-SAT can do it), undoubtedly, by Def. 2.1.

Conclusion:

As demonstrated above, all the four exhaustive possible ways to decide in deterministic polynomial time the Maj2/3-XG-SAT fail: Consequently, there exists a computational decision problem that can be decided in randomized polynomial time, but not in deterministic polynomial time, which implies **P** ≠ **RP**, naturally. Hence, the *derandomization* [15] is a process that does not work in general in our sad computational world. □

For this reason, by union of the Rice's Theorem, the diagonalization method and the complexity classes P and RP, this proof is more a beautiful unification and an amazing

synthesis between the Computability Theory and the Computational Complexity Theory, like the one in [13].

Lastly, someone can say that if a fixed and known $p(n) \geq$ time P(n) of the program **S** into **w** is given (see this one is not deterministic poly-time computable, by Proposition 3.2), then the instances of the Maj2/3-XG-SAT can be reduced to Boolean formula ones by Cook-Levin Theorem, and then if the SAT is decidable in deterministic poly-time, then the Maj2/3-XG-SAT is too. Big idea!

This conclusion is erroneous, however, since knowing a fixed polynomial $p(n) \geq$ that time P(n) is unnecessary to decide the problem (the universal NTM in the proof of Proposition 3.1 and the universal DTM in Way 1.i decide the Maj2/3-XG-SAT without knowing this information (or without such an input), naturally), proving that randomized computation is fundamentally much more faster (even though essentially approximate) than deterministic computation, and that the brute-force or randomized search are unfortunately unavoidable in the real-world computations (I'm very sorry): To verify a correct answer is definitely very easier than find it, naturally.

4.1 Running time of the functions into programs

About running in time P(n) and time greater than P(n), let the function be:

```
01. Poly_Function(string input)
02. {
03.   int i, counter := 0, n := length(input);
04.   for i := 1 to n^10 { counter := counter + 1; } // poly(n) upper bounded running time
05.   if (counter > 100) return(1); else return(0);
06. }
```

The function above evaluated at string input is just a number, naturally. But we can decide that its running time is poly(n) upper bounded, where **n** = |input|. We don't need a TM to decide it. On the other hand, let the function be:

```
01. SuperPoly_Function(string input)
02. {
03.   int i, counter := 0, n := length(input);
04.   for i := 1 to 2^n { counter := counter + 1; } // exp(n) upper bounded running time
05.   if (counter > 100) return(1); else return(0);
06. }
```

Of course, the running time of the function above is exponential in **n**. We know countless functions as the ones above [2] to use them in order to make restricted type X programs. Constructing restricted type X programs using algorithms with known running time is human work, not TM computation [2].

4.2 Example of construction of an instance of the Maj2/3-XG-SAT

Let the restricted type X program **S** be:

```
01. S(string input)
02. {
03.   remainder := mod(integer(input), 4);    // remainder on division of input (converted into
                                               // integer) by 4
```

```
04.   if (remainder < 3) return(Fun2(input)); // returns the value returned by Fun2 and halts
05.   if (remainder = 3) return(Fun1(input)); // never halts
06. }

07. Fun1(string input)
08. {
09.   do { input := "1"; } while (1 = 1); // infinite loop
10.   return(1);
11. }

12. Fun2(string input)
13. {
14.   int i, counter := 0, n := length(input);
15.  for i := 1 to n^10 { counter := counter + 1; } // poly(n) upper bounded running time
16.  if (counter > 0) return(1); else return(0);
17. }
```

Thus, we can simply convert this program **S** into a DTM ***M***, translate it into a binary form **s**, and then construct the well-formed string **w** = **111111110s**, an instance of the Maj2/3-XG-SAT.

Here, hence, constructing Maj2/3-XG-SAT instances, it stands very clear that the human reasoning is much more powerful than mechanical (TM) computation.

5. Baker-Gill-Solovay Theorem and the Proof

Verify that the proof does not use the diagonalization method (except in the justified special cases in Section 4) and it is based about the difference, on worst cases, between running times from a DTM and an NTM (probabilistic TM) that recognize the L_z-language ***L***, as demonstrated in Way 1.i of Section 4 compared to the proof of Proposition 3.1.

Moreover, notice that the addition into the proof methods of oracles to a PSPACE-Complete language *W* does not imply that false statement $P^W \neq NP^W$ (because the proof cannot be adapted to demonstrate that $P^W \neq NP^W$, since a DTM **Q** with an oracle to *W* could simulate any NTM with the same oracle using only a poly(n)-quantity of space, in an adapted Way 1.i, which would otherwise prove that $P^W = NP^W$).

These facts imply that the Baker-Gill-Solovay Theorem of inseparability of the classes P and NP (hence, P and RP) by oracle-invariant methods (techniques that are conserved under the addition of oracles, like the pure diagonalization method without *algebraic oracle* [8]) does not refute this P ≠ RP proof. In other words, my proof technique does not *relativize* [4].

6. Razborov-Rudich Theorem and the Proof

SAT's weakness – The proof does not try to prove any lower bounds on the circuit complexity of a Boolean function, because it does not try to solve the still open question whether SAT is in P, since to prove P ≠ RP it was not necessary to solve the SAT question (for the proof, different from the wrong conclusion in [3, 6], it is irrelevant whether SAT is in P), whereas it was enough to prove that Maj2/3-XG-SAT is in RP but not in P: Thus, the Razborov-Rudich Theorem of the Natural Proofs does not refute this proof. In other words, my proof technique does not *naturalize* [7].

7. Related Work, Aaronson-Wigderson Theorem and the Proof

There is no relevant related work on the goal to really solve the P versus NP question. From important papers upon the matter, there are only some "negative" results, like the ones referred to in Sections 5 and 6 and, more recently, as an extension of the *relativization* in Section 5, the proof that techniques that are conserved under the addition of an oracle and a low-degree extension of it over a finite field or ring cannot work on this question too, by the concept of *algebrization*, explained in [8].

Remember, however, that my proof does not use the pure diagonalization method (as referred to in Section 5), but it exploits properties of computation that are specific to real world computers, and then this new barrier is not valid to refute it, too.

8. Expert Advice & Academic Honesty

A reviewer, referring to the technical report in [17], has said "– It is disconcerting to see how the present author continues to ignore expert advice. His title borders on, and perhaps transgresses, academic honesty. Papers with such grandiose claims should only be considered after an endorsement by an expert."

The heart of my paper is just challenging some traditional definitions on TCS field, essentially the need of polynomial uniformity on the definitions of the complexity classes P and RP. However, that technical report says, for instance: "– As ... Definition 3.5 of his paper ... needs to before the universal quantification on *x* fix a polynomial bounding the length of the certificates, we from here on assume that his definition is viewed as being modified to do that ..."

So, as my proposed new definitions are so distorted in that expert advice, it has very low value in order to evaluate my proof, thus ignoring it is not really academic dishonesty at all, but only logical consequence of that challenge upon enhancing those definitions.

9. Freedom & Mathematics

"**– The essence of Mathematics is Freedom.**" (Georg Cantor) [22]

10. References

[1] J. E. Hopcroft, J. D. Ullman, and R. Motwani, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading MA, 2001.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (Second Edition)*, The Mit Press, Cambridge MA, 2001.

[3] K. J. Devlin, *The Millennium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time*, Basic Books, New York NY, 2002.

[4] M. Sipser, *Introduction to the Theory of Computation – Second Edition*, Thomson Course Technology, Boston MA, 2006.

[5] Adapted from the appendix of the paper *Uniformly Hard Sets* by L. Fortnow and R. Downey, unpublished, available: http://weblog.fortnow.com/media/ladner.pdf

[6] Cook S.A., "*P versus RP problem*", unpublished, available: http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf

[7] From Wikipedia, the free encyclopedia, "*Natural Proof*", unpublished, available: http://en.wikipedia.org/wiki/Natural_proof

[8] S. Aaronson and A. Wigderson, *Algebrization: A New Barrier in Complexity Theory*, Electronic Colloquium on Computational Complexity, Report No. 5 (2008), available: http://eccc.hpi-web.de/eccc-reports/2008/TR08-005/Paper.pdf

[9] O. Goldreich, *On Promise Problems (in memory of Shimon Even (1935-2004))*, unpublished, available: http://www.wisdom.weizmann.ac.il/~oded/PS/prpr.ps

[10] M. Sipser, Cambridge MA 02139, in *The History and Status of the P Versus RP Question*, p. 606, unpublished, available: http://www.seas.harvard.edu/courses/cs121/handouts/sipser-pvsnp.pdf

[11] From Wikipedia, the free encyclopedia, "*Rice's Theorem*", unpublished, available: http://en.wikipedia.org/wiki/Rice's_theorem

[12] O. Goldreich, *Notes on Levin's Theory of Average-Case Complexity*, unpublished, available: http://www.wisdom.weizmann.ac.il/~oded/COL/lnd.pdf

[13] A. L. Barbosa, *P != NP Proof*, unpublished, available: http://arxiv.org/ftp/arxiv/papers/0907/0907.3965.pdf

[14] Chernoff, H. (1952). *A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations*. Annals of Mathematical Statistics 23 (4): 493–507. available: http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177729330

[15] From Wikipedia, the free encyclopedia, "*Randomized Algorithms*", unpublished, available: http://en.wikipedia.org/wiki/Randomized_algorithm - Derandomization

[16] From Wikipedia, the free encyclopedia, "*RP (complexity)*", unpublished, available: http://en.wikipedia.org/wiki/RP_(complexity)

[17] L. A. Hemaspaandra, K. Murray, and X. Tang, *Barbosa, Uniform Polynomial Time Bounds, and Promises*, Technical Report, unpublished, available: http://arxiv.org/abs/1106.1150

[18] T. S. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, Chicago IL, 1962.

[19] From StackExchange (cstheory), cc-wiki, "*Are runtime bounds in P decidable? (answer: no)*", unpublished, available: http://cstheory.stackexchange.com/questions/5004/are-runtime-bounds-in-p-decidable-answer-no

[20] A. L. Barbosa, *NP ⊄ P/poly Proof*, unpublished, available: http://www.andrebarbosa.eti.br/NP_is_not_in_P-Poly_Proof_Eng.pdf

[21] From Wikipedia, the free encyclopedia, "*Constructible Function*", unpublished, available: http://en.wikipedia.org/wiki/Constructible_function

[22] From The Engines of Our Ingenuity, site, "*Episode nº 1484: GEORG CANTOR*", posted by John H. Lienhard, unpublished, available: http://www.uh.edu/engines/epi1484.htm

[23] A. L. Barbosa, *The Cook-Levin Theorem is False*, unpublished, available: http://www.andrebarbosa.eti.br/The_Cook-Levin_Theorem_is_False.pdf

[24] From Gödel's Lost Letter and P=NP, a personal view of the theory of computation, blog, public comments on "*Facts No One Really Checks*", posted at July 25, 2012, by R. J. Lipton, unpublished, available: http://rjlipton.wordpress.com/2012/07/25/facts-no-one-really-checks/#comment-22187

[25] J. Abascal and S. Maimon, *Critique of Barbosa's "P != NP Proof"*, unpublished, available: https://arxiv.org/pdf/1711.07132.pdf

André Luiz Barbosa – Goiânia - GO, Brazil – e-Mail: webmaster@andrebarbosa.eti.br – May 2011

Site.....….. : www.andrebarbosa.eti.br
Blog...….. : blog.andrebarbosa.eti.br

This Paper : http://www.andrebarbosa.eti.br/P_different_RP_Proof_Eng.htm
PDF…..… : http://www.andrebarbosa.eti.br/P_different_RP_Proof_Eng.pdf