

Identifying MFSs with considering masking effect*

[Extended Abstract][†]

Xintao Niu

State Key Laboratory for Novel
Software Technology
Nanjing University
China, 210093

niuxintao@smail.nju.edu.cn

Changhai Nie

State Key Laboratory for Novel
Software Technology
Nanjing University
China, 210093

changhainie@nju.edu.cn

Alvin Chain

Department of computing
Hong Kong Polytechnic
University
Hong Kong

cstschan@comp.polyu.edu.hk

ABSTRACT

Minimal failure-inducing schema(MFS) is a important concept in combinatorial testing that indicate the failure-inducing interaction of parameters in the software under test(SUT). Identify the MFS can help developers quickly reduce the search space needed to find the buggy source. Many algorithms are proposed to find these MFSs in the failing test cases. In practice, however, we find that these algorithms cannot behave as expected in the condition that SUT has multiple faults with different levels. For if so, a fault with higher level may be triggered and leaving the code which will trigger the fault with lower level not executed. Thus we cannot observe the fault with lower level, as a result, we will omit the MFS that related to the fault with lower level. We call this a masking effect.

In this paper, we propose a framework which can help the algorithms to avoid this masking effect when identify MFSs in the test cases. In this framework, we first static analysis the data flow of the test cases. Second we record fault as well as the code lines related to this fault during executing test cases. Then we will determine the levels of different faults we triggered during testing through finding the relationships of the code lines of each fault. By doing so we can judge whether a masking effect is happened during the process of identifying MFS and avoiding them by generating new test case that exposing the fault with lower level. We have applied our framework into several algorithms which focus on identifying the MFS in test cases and empirically studied the framework on two widely used open source software. Our result of the studies shows that our framework can effectively reduce the influence of masking effect.

*(Does NOT produce the permission block, copyright information nor page numbering). For use with ACM-PROC-ARTICLE-SP.CLS. Supported by ACM.

[†]A full version of this paper is available as *Author's Guide to Preparing ACM SIG Proceedings Using L^AT_EX_{2 ϵ} and BibT_EX* at www.acm.org/eaddress.htm

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOODSTOCK '97 El Paso, Texas USA

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous;
D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Minimal failure-inducing schemas, Masking effect

1. INTRODUCTION

With the increase of requirements for more features and customisable, modern software are designed to be configurable and modular. While it can make software portable and flexible, it also bring many challenges to the testers when testing them. The major one of these challenges is that we must make sure the components or options in the software coexistence with each other. As exhaustive testing each possible combination is not impractical when there are large amount of components or options in the SUT, we should choose some of them to test with considering the cost. There are many strategies for one to select the test configurations among all the possible configurations. Combinatorial testing is one of them that can reach the coverage of all the interactions of components with the number of component (we called interaction strength) not more than t .

Which criteria we should choose or how to generate these test cases is not the point in this paper, however, we will focus on the followed problem: if we find some test configurations failed during executing, which subset of the combination of component is the source of this failure? In another word, we want to identify the failure-inducing interactions of component rather than just detect them. Many works (as well as our previous work) are proposed to solve this problem. Most of these works focus on how to identify as more MFSs as possible while just generating small size of extra test configurations.

In our recent studies, however, we find these algorithms cannot behave as expected in some subject software. Through a deep analysis, we find that there are multiple faults with different levels in these subject. It means that when we set up a test configuration and execute the SUT to observe the result, the high level fault will trigger first and perturb we examining the code that may trigger the low level fault. As

a result we will omit some options or component in this test configuration that may be the cause of the low level fault. We call this a masking effect which make the MFS identifying algorithms not able to work properly.

In this paper, we propose a approach that can assist these algorithms to avoid these masking effect. Our framework consists of three parts: first, it will use the statistic analysis technique—dominate tree to analysis the test script and then collect the information traditional identifying algorithms. of code lines in this script. Second, we will support a interface called "Record" for the MFS identifying algorithms that each time the algorithm encounter a fault should call this interface. So that we can record this fault as well as the code lines that trigger this fault. Last, this framework support these algorithms the interface "analysis" that can tell them whether the fault they encounter having masked some fault else.

To evaluate the effectiveness of our framework, we took two widely-used open source software as our experiment subject. And then we will choose five MFSs identifying algorithms, for each algorithm, we will compare the identifying result among two versions of this algorithm, one using our framework while another one not. The result of the empirical studies shows that our framework can assist the MFS identifying algorithm in getting a more accurate result.

The main contributions of this paper are:

1. We show that the fault corresponding to the MFS has different levels, i.e., fault with high level can mask the fault with low level.
2. We give a framework to assist algorithms to avoid the bad influence of the masking effect in different levels of fault.
3. We empirically studies that with considering the masking effect the algorithm can perform better than not considering the effect.

Rest of paper is organised as follows: section 2 gives a simple example to motivate our work. Section 3 describe our framework in detail. Section 4 illustrate the experiment and reports the result. Section 5 discusses the related works. Section 6 provides some concluding remarks.

2. MOTIVATION EXAMPLE

Following we have construct a example to illustrate the motivation of our approach. Assume we have a method *foo* which has four input parameters : *a*, *b*, *c*, *d*. The types of these four parameters are all integers and the values that they can take are: $d_a = \{7, 11\}$, $d_b = \{2, 4, 5\}$, $d_c = \{4, 6\}$, $d_d = \{3, 5\}$ respectively. The detail code of this method is listed as following:

```
public static float foo(int a, int b, int c, int d){
    //step 1 will cause a exception when b == c
    float x = (float)a / (b - c);

    //step 2 will cause a exception when c < d
    float y = Math.sqrt(c - d);

    return x+y;
}
```

Table 1: test inputs and their corresponding result

id	test inputs	result
1	(7, 2, 4, 3)	PASS
2	(7, 2, 4, 5)	Ex 2
3	(7, 2, 6, 3)	PASS
4	(7, 2, 6, 5)	PASS
5	(7, 4, 4, 3)	Ex 1
6	(7, 4, 4, 5)	Ex 1
7	(7, 4, 6, 3)	PASS
8	(7, 4, 6, 5)	PASS
9	(7, 5, 4, 3)	PASS
10	(7, 5, 4, 5)	Ex 2
11	(7, 5, 6, 3)	PASS
12	(7, 5, 6, 5)	PASS
13	(11, 2, 4, 3)	PASS
14	(11, 2, 4, 5)	Ex 2
15	(11, 2, 6, 3)	PASS
16	(11, 2, 6, 5)	PASS
17	(11, 4, 4, 3)	Ex 1
18	(11, 4, 4, 5)	Ex 1
19	(11, 4, 6, 3)	PASS
20	(11, 4, 6, 5)	PASS
21	(11, 5, 4, 3)	PASS
22	(11, 5, 4, 5)	Ex 2
23	(11, 5, 6, 3)	PASS
24	(11, 5, 6, 5)	PASS

Inspecting the simple code above, we can find two faults: First, in the step 1 we can get a `ArithmeticException` when *b* is equal to *c*, i.e., $b = 4$ & $c = 4$, that makes division by zero. Second, another `ArithmeticException` will be triggered in step 2 when $c < d$, i.e., $c = 4$ & $d = 5$, which makes square roots of negative numbers. So the expected MFSs in this example should be $(-, 4, 4, -)$ and $(-, -, 4, 5)$.

Traditional MFS identifying algorithms do not consider the detail of the code. They take black-box testing of this program, i.e., feed inputs to those programs and execute them to observe the result. The basic justification behind those approaches is that the failure-inducing schema for a particular fault must only appear in those inputs that trigger this fault. As traditional MFS identifying algorithms aim at using as small number of inputs as possible to get the same or approximate result as exhaustive testing, so the results derive from a exhaustive testing set must be the best that these MFS identifying approaches can reach. Next we will illustrate how exhaustive testing works on identifying the MFS in the program.

We first generate every possible inputs as listed in the Column "test inputs" of table 1, and execute them to get the result listed in Column "result" of table 1. In this Column, "PASS" means that the program runs without any exception under the inputs in the same row. "Ex 1" indicate that the program encounter a exception corresponding to the step 1 and "Ex 2" indicate the program trigger a exception corresponding to the step 2. According to data listed in table 1, we can deduce that that $(-, 4, 4, -)$ must be the MFS of Ex 1 as all the inputs triggered Ex 1 contain this schema. Similarly, the schema $(-, 2, 4, 5)$ and $(-, 3, 4, 5)$ must be the MFSs of the Ex 2. We listed the MFSs and its corresponding exception in table 2.

Table 2: Identified MFSs and their corresponding Exception

MFS	Exception
(-, 4, 4, -)	Ex 1
(-, 2, 4, 5)	Ex 2
(-, 3, 4, 5)	Ex 2

Table 3: expected MFSs and their corresponding Exception

MFS	Exception
(-, 4, 4, -)	Ex 2
(-, -, 4, 5)	Ex 1

Note that we didn't get the expected result with traditional MFS identifying approaches for this case. The MFSs we get for Ex 2 are (-,2,4,5) and (-,3,4,5) respectively instead of the expected schema (-,-,4,5). So why we can't identify the MFS (-,-,4,5)? The reason lies in the two inputs: input 6. (7,4,4,5) and input 18. (11,4,4,5). This two inputs contain the schema (-,-,4,5), but didn't trigger the Ex 1, instead, the Ex 2 was triggered.

Now let us get back to the source code of *foo*, we can find that if Ex 1 are triggered, it will stop executing the remaining code and report the exception information. In another word, Ex 1 have a higher level than Ex 2 so that Ex 1 may mask Ex 2. With this information, we can suppose that for the input (7,4,4,5) and (11,4,4,5), Ex 1 may masked Ex 2. Then we exam the schema (-,-,4,5), we can find all the test inputs contain (-,-,4,5) will trigger exception 1, except these test cases trigger Ex 2 first. So we can conclude that (-,-,4,5) should be the causing schema of the Exception 1. So the MFS information will be updated to table 3 which is identical to the expected result.

So in this paper, we need to analysis the priority among the faults in the SUT and use the information to assist the MFS identifying algorithms to make the result more accurate and clearer.

3. PRELIMINARY

Before we talk about our approach, we will give some formal definitions and background first, which is helpful to understand the description of our approach.

3.1 Combinatorial testing

Assume that the SUT (software under test) is influenced by n parameters, and each parameter c_i has a_i discrete values from the finite set V_i , i.e., $a_i = |V_i|$ ($i = 1, 2, \dots, n$). Some of the definitions below are originally defined in .

Definition 1. A *test configuration* of the SUT is an array of n values, one for each parameter of the SUT, which is denoted as a n -tuple (v_1, v_2, \dots, v_n) , where $v_1 \in V_1, v_2 \in V_2 \dots v_n \in V_n$.

Definition 2. We consider the fact that abnormal executing of the SUT as a *fault*. It can be a exception, a compilation error, a mismatched assertion or a constraint violation.

Definition 3. The *priority* is a function indicate the priority relationship between two faults. Specifically, we take

$Priority(F_a, F_b) = 1$ as that fault F_a has a higher level than F_b , which means that if F_a were triggered, it will omit the code that may trigger F_b . And $Priority(F_a, F_b) = -1$ indicate that F_a has a lower level than F_b . Finally, we take $Priority(F_a, F_b) = 0$ as that there is no priority relationship between F_a and F_b , in another word, neither F_a will mask F_b nor F_b will mask F_a .

Definition 4. For the SUT, the n -tuple $(-, v_{n_1}, \dots, v_{n_k}, \dots)$ is called a k -value *schema* ($k > 0$) when some k parameters have fixed values and the others can take on their respective allowable values, represented as "-". In effect a test configuration its self is a k -value *schema*, which k is equal to n . Furthermore, if a test configuration contain a *schema*, i.e., every fixed value in this schema is also in this test configuration, we say this configuration hit this *schema*.

Definition 5. let s_l be a l -value schema, s_m be an m -value schema for the SUT and $l \leq m$. If all the fixed parameter values in s_l are also in s_m , then s_m *subsumes* s_l . In this case we can also say that s_l is a *sub-schema* of s_m and s_m is a *parent-schema* of s_l .

Definition 6. If all test configurations except these configurations triggered a higher level fault contain a schema, say S_a , trigger a particular fault, say F_a , then we call this schema S_a the *faulty schema* for F_a . Additionally, if none sub-schemas of S_a is the *faulty schema* for F_a , we will call the schema S_a the *minimal faulty schema* for F_a (MFS for short).

Note that, traditional MFS definition didn't consider the priority relationship among faults, so these definition will not take the schema as a MFS for some particular fault if some test configuration contain this schema doesn't trigger this fault.

The target of traditional MFS identifying algorithms is to find the MFSs for the faults of a SUT. For by doing that can help the developers reduce the scope of source code that needed to inspect to debug the software. Note that our discuss is based on the SUT is a deterministic software, i.e., SUT execute under a test configuration will not pass one time and fail another time. The non-deterministic problem will complex our test scenario, which, however is beyond the scope of this paper.

3.2 Dominate tree

fault. level. given this definition, we can also take the constraint as a fault, usually it will have the highest level, for that if we take a combination which is constraint, it will may didn't compile at all, so that we can't exam any code in this software.

test configuration?

schema. faulty, healthy. we need to identify the minimal faulty schema, that is MFS.

note that in our paper, we just consider the failure that is deterministic.

4. APPROACH DESCRIPTION

In this section, we will give a detail description of our approach and a simple example will be attached to illustrate this algorithm.

Traditional MFS identifying algorithms all including a step: fix some parts of the original failing test configuration and then mutant other parts to generate newly test

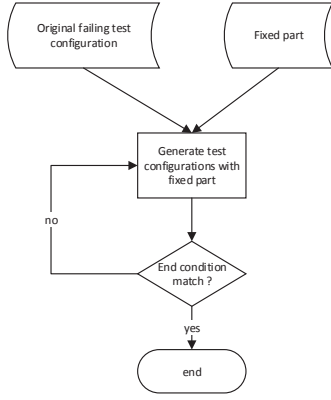


Figure 1: the overview of our approach.

configuration. This step is the basis of these algorithms, as it can help them to compare other configurations, to isolate the failure-inducing parts, to help rank the possible schemas and so on. Our approach is to improve the quality of generated test configuration, so that to assist these algorithm to reduce the influence of the faults masking effects.

As a overview of our approach, our approach is listed as figure 4:

This figure tell us when given a original failing test configuration and the fixed part, our approach will repeat generating test configurations which mutant the elements in the original test configuration except contain this fixed part until the end condition is matched.

1. Mutant other parts of the original test configurations
2. Execute the SUT under the test configuration make some according to the result.
original failing test configuration. and fixed part.
generate test configurations.
judge the based on reuslt
end. regenerate.

In a word, our approach takes a revalidation strategy when encounter different faults until we find a test case. Next we will discuss the two parts in detail.

4.1 judge the result

There are three condition we will end our algorithm:

1. If the generated test configuration passed during testing. This means that the fixed part we validated does not contain the failure-inducing elements, so we will send a message to tell the MFS identifying algorithm that this fixed part is a healthy part, and return this passing generated test configuration to these algorithms.
- 2.If the generated test configuration failed with the same faulty information as the original test configuration.
- 3.We cannot generate anymore test configurations contained this fixed part and diff from the original test configuration.

In this case, we didn't know which condition does this fixed part should be because we didn't find any test configuration that contain this part and either passed or failed with the same fault information. In this paper, we default set this fixed part as the failure-inducing . And return any test configuration generated.

In fact, to be efficiency, we set the repeat times as a fixed number, say 3 to be end as soon as possible.

Reversely, the only condition that we did not stop is that we encounter a different fault information as the original test configuration, for in that case we did not has enough information whether this fixed part is a failure-inducing part or not as the different fault information may be a masking effect.

4.2 regenerate

The regenerate step is to regenerate test configuration that contain the fixed part and does not contain any same elements in the remained part. The reason is that if so we can't not make sure the fixed part or the contained remained part is the failure-inducing part.

Further more, as we having set a fixed number, we can't running all the possible test configuration, so if we did not encounter the passed or the same fault, we will take some more test configuration instead all of them to test. As similar test configuration will trigger the same fault, so we vary them as different as possible to avoid the following condition:

all the regenerated test configuration trigger a fault that have a higher level that the original one. The method we generate the test configuration it using random method. that is, using random number to take these unfixed part.

We generate as different as possible test cases for the tuple, to avoid some more MFSs that may mask this tuple. To do this we maintain a hash table to record the tuple and its corresponding test cases generated, each time we generate a random test case for this tuple, we will lookup this table to find if this test case is already generated, thus we will avoid generating redundancy test cases.

we will give simple example next to illustrate our approach. Take the fic as a example.

4.3 simple example

The approach, our approach is that,
if the algorithm want to get the result, we will put off the result, and generate another one.

generate test cases:

Five setup as follows:

Another note is that:

In detail, for FIC, TRT, OFOT, it can just use the original approach, that it each time test one tuple, and the we use our framework to assign this task.

For CTA, we can use our framework to choose a set of test cases for it to identify the MFS, like shykara do, but different from her work, we just not one test one time, we diff the test cases for different fault.

For SP, we can use our framework for him to generate test cases, it will test multiple tuples one time, and we just first for one fault, then another.

So our comparison work is just compare the one fault, with a noise for the multiple faults.

5. EMPIRICAL STUDIES

To evaluate our approach, we take two open-source software as our subjects: HSQLDB and . HSQLDB is a pure-java data-base management software, and is a . Both of them have a large support developers' forum. Further more as our approach is a framework that can bee seen as a for the MFS identifying algorithms, so that we afford them five algorithms: fic ofot trt cta sp.

There are three questions we want to figure out in this empirical studies:

Q1: does our framework reduce the number of MFSs and the degree of MFS?

Q2: does our framework can validate that MFSs exist masking effect.

Q3: does our framework the result is match the result in developer's forum.

5.1 experiment set up

5.2 evaluation metrics

5.3 result and analysis

6. RELATED WORKS

combinatorial testing has many factors, Nie give a survey, at some are focus.

Identify MFS are

then consider the masking effect is, to my best knowledge, only one paper, unfortunately, it just consider the.

Yilmaz propose a work that is feedback driven combinatorial testing, different from our work, it first using CTA classify the possible MFS and then eliminate them and generate new test cases to detect possible masked interaction in the next iteration. The difference is that the main focus of that work is to generate test cases that didn't omit some schemas that may be masked by other schemas. And our work is main focus on identifying the MFS and avoiding the masking effect.

7. CONCLUSION

8. THE BODY OF THE PAPER

Typically, the body of a paper is organized into a hierarchical structure, with numbered or unnumbered headings for sections, subsections, sub-subsections, and even smaller sections. The command `\section` that precedes this paragraph is part of such a hierarchy.¹ L^AT_EX handles the numbering and placement of these headings for you, when you use the appropriate heading commands around the titles of the headings. If you want a sub-subsection or smaller part to be unnumbered in your output, simply append an asterisk to the command name. Examples of both numbered and unnumbered headings will appear throughout the balance of this sample document.

Because the entire article is contained in the `document` environment, you can indicate the start of a new paragraph with a blank line in your input file; that is why this sentence forms a separate paragraph.

8.1 Type Changes and Special Characters

We have already seen several typeface changes in this sample. You can indicate italicized words or phrases in your text with the command `\textit`; emboldening with the command `\textbf` and typewriter-style (for instance, for computer code) with `\texttt`. But remember, you do not have

¹This is the second footnote. It starts a series of three footnotes that add nothing informational, but just give an idea of how footnotes work and look. It is a wordy one, just so you see how a longish one plays out.

to indicate typestyle changes when such changes are part of the *structural* elements of your article; for instance, the heading of this subsection will be in a sans serif² typeface, but that is handled by the document class file. Take care with the use of³ the curly braces in typeface changes; they mark the beginning and end of the text that is to be in the different typeface.

You can use whatever symbols, accented characters, or non-English characters you need anywhere in your document; you can find a complete list of what is available in the *L^AT_EX User's Guide*[?].

8.2 Math Equations

You may want to display math equations in three distinct styles: inline, numbered or non-numbered display. Each of the three are discussed in the next sections.

8.2.1 Inline (In-text) Equations

A formula that appears in the running text is called an inline or in-text formula. It is produced by the `math` environment, which can be invoked with the usual `\begin. . .\end` construction or with the short form `$. . . $`. You can use any of the symbols and structures, from α to ω , available in L^AT_EX[?]; this section will simply show a few examples of in-text equations in context. Notice how this equation: $\lim_{n \rightarrow \infty} x = 0$, set here in in-line math style, looks slightly different when set in display style. (See next section).

8.2.2 Display Equations

A numbered display equation – one set off by vertical space from the text and centered horizontally – is produced by the `equation` environment. An unnumbered display equation is produced by the `displaymath` environment.

Again, in either environment, you can use any of the symbols and structures available in L^AT_EX; this section will just give a couple of examples of display equations in context. First, consider the equation, shown as an inline equation above:

$$\lim_{n \rightarrow \infty} x = 0 \tag{1}$$

Notice how it is formatted somewhat differently in the `displaymath` environment. Now, we'll enter an unnumbered equation:

$$\sum_{i=0}^{\infty} x + 1$$

and follow it with another numbered equation:

$$\sum_{i=0}^{\infty} x_i = \int_0^{\pi+2} f \tag{2}$$

just to demonstrate L^AT_EX's able handling of numbering.

8.3 Citations

Citations to articles [?, ?, ?, ?], conference proceedings [?] or books [?, ?] listed in the Bibliography section of your article will occur throughout the text of your article. You should use BibTeX to automatically produce this bibliography; you simply need to insert one of several citation

²A third footnote, here. Let's make this a rather short one to see how it looks.

³A fourth, and last, footnote.

Table 4: Frequency of Special Characters

Non-English or Math	Frequency	Comments
Ø	1 in 1,000	For Swedish names
π	1 in 5	Common in math
\$	4 in 5	Used in business
Ψ_1^2	1 in 40,000	Unexplained usage

commands with a key of the item cited in the proper location in the `.tex` file [?]. The key is a short reference you invent to uniquely identify each work; in this sample document, the key is the first author’s surname and a word from the title. This identifying key is included with each item in the `.bib` file for your article.

The details of the construction of the `.bib` file are beyond the scope of this sample document, but more information can be found in the *Author’s Guide*, and exhaustive details in the *L^AT_EX User’s Guide*[?].

This article shows only the plainest form of the citation command, using `\cite`. This is what is stipulated in the SIGS style specifications. No other citation format is endorsed or supported.

8.4 Tables

Because tables cannot be split across pages, the best placement for them is typically the top of the page nearest their initial cite. To ensure this proper “floating” placement of tables, use the environment `table` to enclose the table’s contents and the table caption. The contents of the table itself must go in the `tabular` environment, to be aligned properly in rows and columns, with the desired horizontal and vertical rules. Again, detailed instructions on `tabular` material is found in the *L^AT_EX User’s Guide*.

Immediately following this sentence is the point at which Table 1 is included in the input file; compare the placement of the table here with the table in the printed dvi output of this document.

To set a wider table, which takes up the whole width of the page’s live area, use the environment `table*` to enclose the table’s contents and the table caption. As with a single-column table, this wide table will “float” to a location deemed more desirable. Immediately following this sentence is the point at which Table 2 is included in the input file; again, it is instructive to compare the placement of the table here with the table in the printed dvi output of this document.

8.5 Figures

Like tables, figures cannot be split across pages; the best placement for them is typically the top or the bottom of the page nearest their initial cite. To ensure this proper “floating” placement of figures, use the environment `figure` to enclose the figure and its caption.

This sample document contains examples of `.eps` and `.ps` files to be displayable with L^AT_EX. More details on each of these is found in the *Author’s Guide*.

As was the case with tables, you may want a figure that spans two columns. To do this, and still to ensure proper “floating” placement of tables, use the environment `figure*` to enclose the figure and its caption. and don’t forget to end the environment with `figure*`, not `figure`!

Note that either `.ps` or `.eps` formats are used; use the



Figure 2: A sample black and white graphic (.eps format).



Figure 3: A sample black and white graphic (.eps format) that has been resized with the epsfig command.

`\epsfig` or `\psfig` commands as appropriate for the different file types.

8.6 Theorem-like Constructs

Other common constructs that may occur in your article are the forms for logical constructs like theorems, axioms, corollaries and proofs. There are two forms, one produced by the command `\newtheorem` and the other by the command `\newdef`; perhaps the clearest and easiest way to distinguish them is to compare the two in the output of this sample document:

This uses the `theorem` environment, created by the `\newtheorem` command:

THEOREM 1. *Let f be continuous on $[a, b]$. If G is an antiderivative for f on $[a, b]$, then*

$$\int_a^b f(t)dt = G(b) - G(a).$$

The other uses the `definition` environment, created by the `\newdef` command:

Definition 7. If z is irrational, then by e^z we mean the unique number which has logarithm z :

$$\log e^z = z$$

Two lists of constructs that use one of these forms is given in the *Author’s Guidelines*.

There is one other similar construct environment, which is already set up for you; i.e. you must *not* use a `\newdef` command to create it: the `proof` environment. Here is an example of its use:

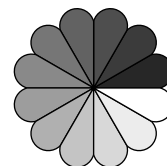


Figure 5: A sample black and white graphic (.ps format) that has been resized with the psfig command.

Table 5: Some Typical Commands

Command	A Number	Comments
<code>\alignauthor</code>	100	Author alignment
<code>\numberofauthors</code>	200	Author enumeration
<code>\table</code>	300	For tables
<code>\table*</code>	400	For wider tables

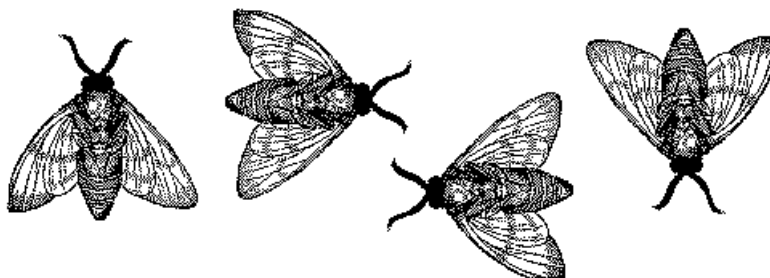


Figure 4: A sample black and white graphic (.eps format) that needs to span two columns of text.

PROOF. Suppose on the contrary there exists a real number L such that

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = L.$$

Then

$$l = \lim_{x \rightarrow c} f(x) = \lim_{x \rightarrow c} \left[g(x) \cdot \frac{f(x)}{g(x)} \right] = \lim_{x \rightarrow c} g(x) \cdot \lim_{x \rightarrow c} \frac{f(x)}{g(x)} = 0 \cdot L = 0,$$

which contradicts our assumption that $l \neq 0$. \square

Complete rules about using these environments and using the two different creation commands are in the *Author's Guide*; please consult it for more detailed instructions. If you need to use another construct, not listed therein, which you want to have the same formatting as the Theorem or the Definition[?] shown above, use the `\newtheorem` or the `\newdef` command, respectively, to create it.

A Caveat for the T_EX Expert

Because you have just been given permission to use the `\newdef` command to create a new form, you might think you can use T_EX's `\def` to create a new command: *Please refrain from doing this!* Remember that your L^AT_EX source code is primarily intended to create camera-ready copy, but may be converted to other forms – e.g. HTML. If you inadvertently omit some or all of the `\defs` recompilation will be, to say the least, problematic.

9. CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the L^AT_EX book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only.

10. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the `.cls` and `.tex` files that it describes.

APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the `appendix` environment, the command `section` is used to indicate the start of each Appendix, with alphabetic order designation (i.e. the first is A, the second B, etc.) and a title (if you include one). So, if you need hierarchical structure *within* an Appendix, start with `subsection` as the highest level. Here is an outline of the body of this document in Appendix-appropriate form:

A.1 Introduction

A.2 The Body of the Paper

A.2.1 Type Changes and Special Characters

A.2.2 Math Equations

Inline (In-text) Equations.

Display Equations.

A.2.3 Citations

A.2.4 Tables

A.2.5 Figures

A.2.6 Theorem-like Constructs

A.3 Conclusions

A.4 Acknowledgments

A.5 Additional Authors

This section is inserted by L^AT_EX; you do not insert it. You just add the names and information in the `\additionalauthors` command at the start of the document.

A.6 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

B. MORE HELP FOR THE HARDY

The sig-alternate.cls file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of L^AT_EX, you may find reading it useful but please remember not to change it.