

An Automated Test Evaluation Tool

The AutoCheck program fully automates the evaluation of test protocol files for medical patient monitors. The AutoCheck output documents that the evaluation has been carried out and presents the results of the evaluation.

by Jörg Schwering

The AutoCheck program extends the automated test environment described in this journal in 1991.¹ It fully automates the evaluation of the test protocol files generated by the AutoTest program.

Fig. 1 is a brief summary of the 1991 system. The main part of the figure shows the main elements of the AutoTest program and its environment. The AutoTest program reads and interprets commands line-by-line out of a test script (a test procedure). Basically there are three types of commands:

- Commands to simulate user input on the monitor under test (keypusher)
- Commands to control the signal simulators, which play the role of a critically ill patient
- Commands to request and log output data from the monitor under test (reactions to keypresses and signals applied).

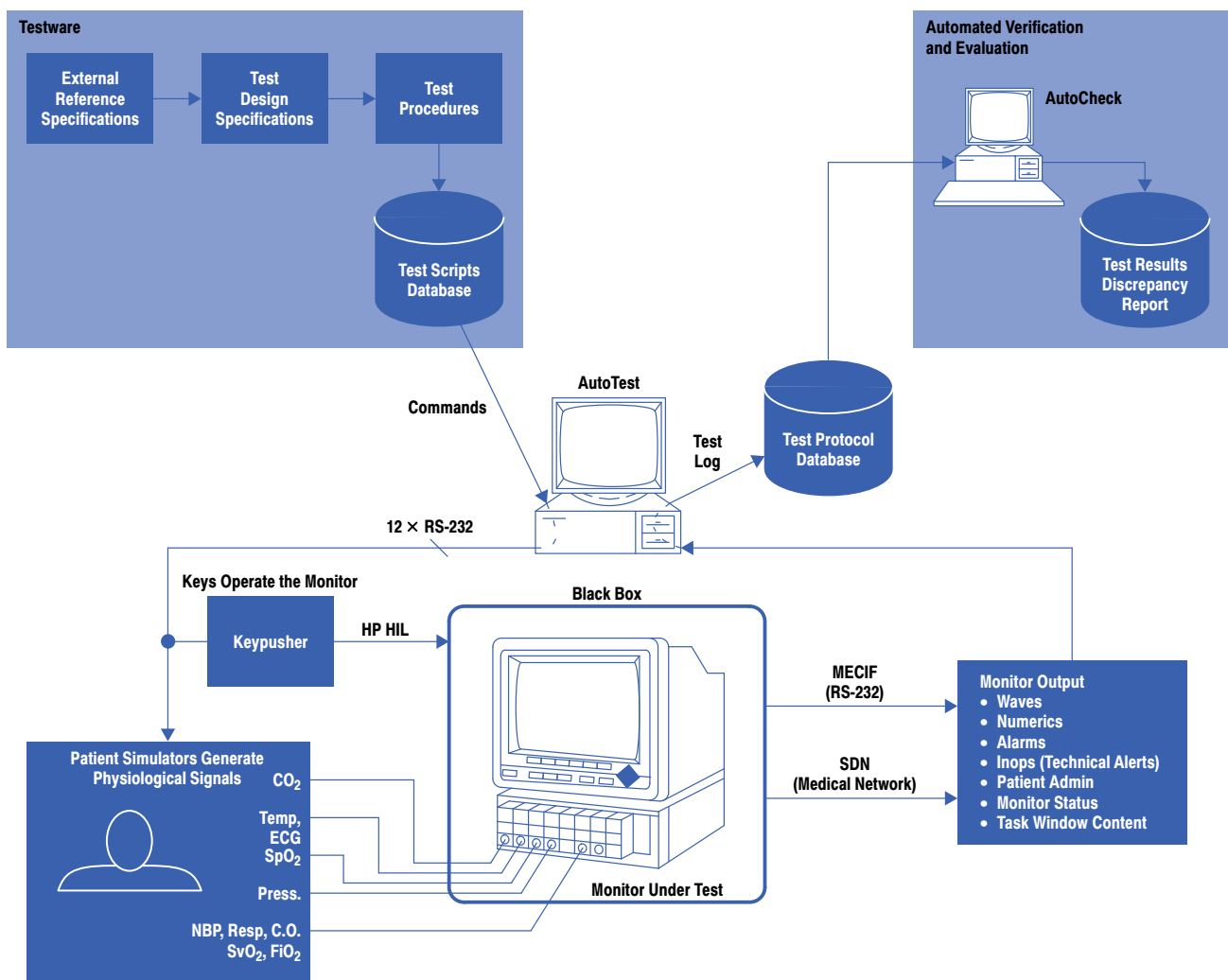


Fig. 1. Automated test environment for medical patient monitoring systems. AutoCheck, the automatic verification and evaluation tool shown in the upper right corner, is a recent addition.

that needed to be repeated, the evaluation was normally restricted to a few (one to three) selected repetitions. Statistical tests, such as adjusting an alarm limit randomly and checking the alarm string, generate particularly large protocol files that are difficult to evaluate manually, leading the test engineer to reduce the number of test cases to a minimum.

Goals for an Automatic Test Evaluation Tool

Because of these problems, an investigation was started on a tool that could replace the human evaluator. The goals for this automatic evaluation tool were:

- To relieve the test engineer of tedious, time-consuming manual evaluation and thereby increase efficiency
- To avoid overlooking discrepancies
- To get the test results faster by quicker evaluation
- To increase test coverage through side-effect checks
- To make evaluation more objective (not tester dependent)
- To allow conditional checks (flexibility)
- To automate local language regression tests (see [Article 15](#)).

Use Model

The basic use model (see Fig. 3) is the replacement of manual evaluation with the automatic evaluation tool. The evaluation of the protocol file runs after the test has finished. The test files already contain the expected results coded in a formal language readable by AutoCheck.

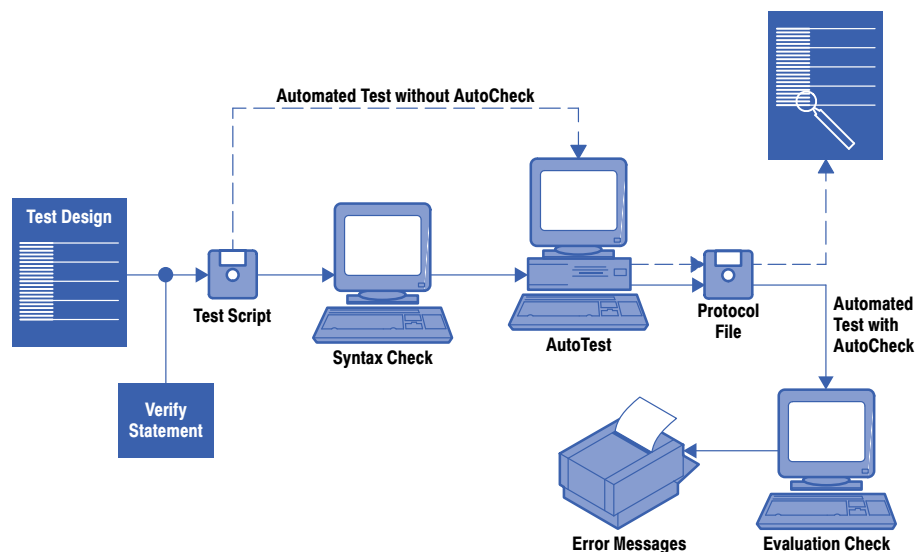


Fig. 3. The use model for AutoCheck replaces manual evaluation with the automatic evaluation tool.

Test execution and evaluation now consists of the following steps:

1. Write the AutoTest test script including the expected results in AutoCheck format. The basic test script layout as described above stays the same. The only differences are that some AutoCheck definitions such as tolerances (see “AutoCheck Features and Syntax” below) are added to the startup block and that the description of the expected results has to follow the AutoCheck format.
2. Run this test script through the AutoCheck syntax check to avoid useless AutoTest runs.
3. Execute the test script with AutoTest as usual. The expected results (AutoCheck statements) are treated by AutoTest as comments, which means that they are only copied into the protocol file together with a time stamp.
4. Run the protocol file through the AutoCheck evaluation check, which includes a syntax check. AutoCheck generates a diff file reporting the deviations from the expected results (errors) and warnings for everything that couldn't be evaluated or is suspicious in some other way (for details see “AutoCheck Output” below).
5. If and only if AutoCheck reports errors or warnings, check the protocol file to find out whether the deviation is caused by a flaw in the test script or a bug in the patient monitor under test.

Architecture

We first conducted a feasibility study, which investigated different architectural approaches and implementation tools. The first approach was in the area of artificial intelligence, namely expert systems and language recognition (this would be expected for an investigation started in 1991). It soon became apparent that protocol file evaluation is basically a compiler problem. The languages and tools investigated were Prolog/Lisp, sed/UNIX[®] shell, lex/yacc, C, and a C-style macro language for a programmable editor. We came to the conclusion that a combination of lex/yacc and C would lead to the easiest and most flexible solution.

Fig. 4 shows the AutoCheck architecture. The protocol file is first run through a preprocessor, which removes all lines irrelevant to AutoCheck, identifies the different AutoTest interfaces, and performs the local language translations. Thereafter it is analyzed by a combination of a scanner and a parser. We implemented specialized scanner/parsers for the AutoCheck metalanguage and the data provided by the different patient monitor interfaces. The AutoCheck statements and the AutoTest data are written into separate data structures. A third data structure holds some control parameters such as the accepted tolerances (see "AutoCheck Features and Syntax" below). After each data package, which is the answer to one AutoTest data request command, the compare function is started. The compare function writes all deviations into the error file.

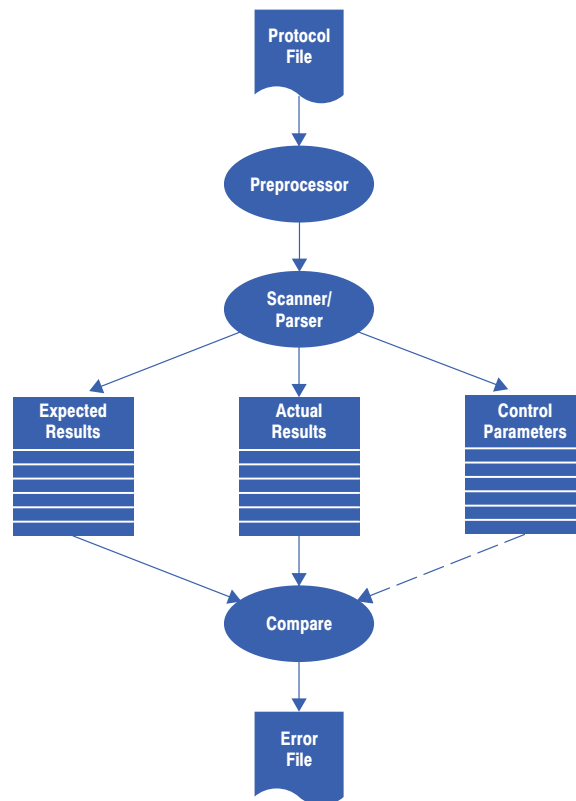


Fig. 4. AutoCheck architecture.

Basically, AutoTest and AutoCheck recognize two types of data requests: *single tunes*, which respond with exactly one data set for each requested message, and *continuous tunes*, which gather data over a defined time interval.

In the monitor family under test all important data has an update frequency of 1024 ms. AutoTest groups all data messages received within a 1024-ms frame into one data block and AutoCheck treats each data block of a continuous tune like a data package of a single tune.

All AutoCheck statements are then verified against each data block. The AutoCheck statements remain valid and are applied to the next data block until they are explicitly cleared or overwritten by a new AutoCheck block.

AutoCheck Features and Syntax

The AutoCheck features and syntax are best described using the example shown in Fig. 5. The numbers below correspond to the numbers in the left column of Fig. 5.

1. All AutoCheck statements are preceded by a caret (^) and are treated as comments by AutoTest. As mentioned above under "Architecture," AutoCheck statements are grouped into blocks. Each block is enclosed by the two statements `Verify Begin` and `Verify End`.

```

(4) ^ Tolerance Definition
(4) ^   "Temp1" : 1%;
(4) ^ End Tolerance Definition

(1) ^ Verify Begin
(2)(4) ^   "Temp1"->value = 37.0;
(2) ^   "Temp1"->unit = C;
(3) ^   not alarm for "Press1"
(3) ^     within (5,NaN);
(5) ^   alarm "HR" > al_max;
(6)(7) ^   if Neonate
(6) ^     then
(6) ^       "HR"-> al_min = 30;
(6) ^     endif;
(6)(7) ^   if value of "Pat.Size" is "Adult"
(6) ^     then
(6) ^       "HR"-> al_min = 15;
(6) ^     endif;
(8) ^   write "Check user input";
(1) ^ Verify End)

```

Fig. 5. An example of expected results written in the AutoCheck language. The numbers at the left refer to the paragraphs in the article that describe these statements.

2. There is a set of AutoCheck statements that enables the user to verify all data that can be read by AutoTest (numerics, alerts, sound, wave data, task window texts, etc.). An example of a numerical value is temperature, including all of its accompanying attributes such as units (°C) and alarm limits. In this example the value of the temperature numeric is expected to be 37.0°C.
3. Verify statements can be combined with:
 - A negation, for example to check the absence of an alarm
 - Timing conditions, for example to verify that an alarm delay is within its specified range.

In this example it is expected that in the time interval from 5 seconds to infinity (NaN) there is no alarm for blood pressure. This is a typical test case in which there was an alarm and the simulated measurement has been reset between the alarm limits, the object being to check that the alarm disappears within a defined time.
4. For all numerical values (measurements), including those in the alarm string, a tolerance can be defined to compensate for simulator tolerances. The tolerances are defined outside the Verify block in an additional block. Although the user can change the tolerances as often as desired, they are typically defined once at the beginning of a test procedure and then used for the whole test procedure. In this example, all values in the range from 1% below 37.0°C to 1% above 37.0°C (36.7 to 37.3°C) would be accepted as correct for the Temp1 parameter.
5. There are special combinations, such as a numeric value and an alarm string. For instance, in the monitor family under test an alarm message typically indicates that the alarm limit has been exceeded. The alarm limit is also included in a numeric message along with its attributes. The command alarm "HR" > al_max allows the tester to compare the alarm limit in the alarm message with the alarm limit in the numeric message (as opposed to checking both messages against a fixed limit). This feature is mainly useful for statistical tests.
6. Simple control structures (if, and, or) can be used to define different expected results for conditions that are either not controllable by the test environment or are deliberately not explicitly set to expand the test coverage. In the monitor family under test some settings are dependent on the configuration (e.g., patient size). The simple control structures allow configuration-dependent evaluation
7. As a condition in an if statement, either flags, which have to be defined earlier in the test procedure, or an ordinary AutoCheck statement can be used.
8. AutoCheck provides a command to write a comment into the output file. This can be used to instruct the user to check something manually (e.g., a user input string).

AutoCheck Output

Fig. 6 is an example of AutoCheck output. AutoCheck generates seven different output types:

- Evaluation Error. The expected data and the received data don't match.

- Evaluation Warning. AutoCheck couldn't determine whether the data is correct (e.g., data missing).
- AutoTest Error. Errors reported by AutoTest are mirrored in the output file to make them visible to the test engineer, who only looks at the protocol file in case of reported errors.
- Syntax Error. The interpretation of the AutoCheck syntax failed.
- Syntax Warning. The AutoCheck syntax could be interpreted, but is suspected to be incomplete or wrong.
- Data Error. The AutoTest data couldn't be interpreted correctly. This indicates either a corrupted protocol file or an incompatibility of AutoCheck and AutoTest versions.
- Write. This is a user-defined output. It enables the user to mark data that should be checked manually, such as user input at a pause statement.

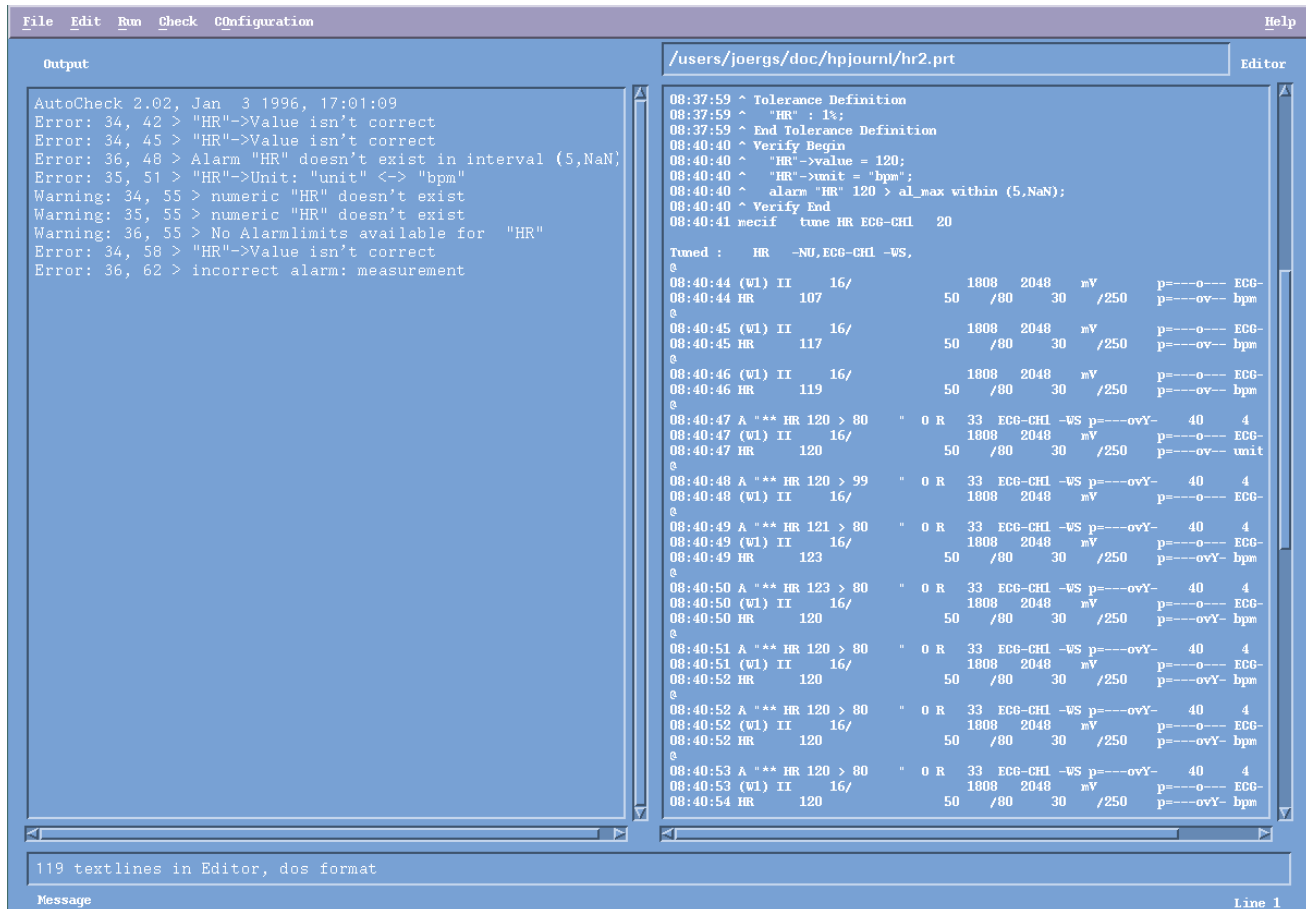


Fig. 6. An example of AutoCheck output.

The user can choose between four different warning levels for the syntax and evaluation warnings and can switch individual warnings on or off.

The output generated by AutoCheck has the following format:

```

ErrorType: statementline, dataline >
           descriptive_text

```

Thus, both the line containing the AutoCheck statement and the line containing the data are indicated.

If the output is written into a file, each line is preceded by the filename(statementline). This is the same format as used by many compilers, and therefore the built-in macros of many programming editors can be used in combination with AutoCheck. This means that errors can be reviewed in much the same way that a source file is debugged after compilation using an editor pointing to the source code errors.

At the end of the evaluation, AutoCheck gives the test engineer a quick overview of the result by providing a table showing how many output messages of each type have been generated. Whereas the evaluation errors indicate bugs either in the product or in the test script, the other output messages indicate potential problems in the test execution or evaluation process.

The AutoCheck output documents both that the evaluation has been carried out and the result of the evaluation, which for medical products are important for regulatory approvals and audits.

Platforms

AutoCheck and Autotest run on different platforms. AutoTest runs on a DOS[®]-based PC, which is very appropriate as a test controller because of the inexpensive hardware, an operating system that doesn't require dealing with tasking conflicts, and the availability of interface cards (the interface to the medical network is available as a PC card only). AutoCheck runs on a UNIX-based workstation because of the availability of *lex/yacc* and the greater resources (memory and processing power). However, both tools work on the same file system (a UNIX-based LAN server). The user doesn't have to worry about the different file formats, because AutoCheck automatically takes care of the format conversions. It accepts both DOS and UNIX formats and generates the output according to the detected protocol file format. Having different machines for execution and evaluation has also not proved to be a disadvantage for the test engineer.

Expandability

The basic architecture of AutoCheck has proven to be flexible for enhancements over time. Since the first release of AutoCheck we have implemented many enhancements because of new product features and because AutoTest provides additional data structures.

Validation

The risk of the AutoCheck approach is that, if AutoCheck overlooks an error (false negative output), the tester won't find the error. An automatic evaluation tool is only useful if the tester can rely on it, since otherwise, even if no errors were reported, the tester would still have to look at the protocol file. Therefore, the validation of an automatic evaluation tool is crucial to the success of such a tool. For this reason a thorough test of the tool was designed and every new revision is regression tested. Changes and enhancements undergo a formal process similar to that used for customer products.

Results

The manual evaluation time for an overnight test of around one to two hours has been reduced by the use of AutoCheck to less than a minute. This means that the additional effort for the test engineer for writing the expected results in the AutoCheck syntax is compensated after three to five test runs. This depends on the experience of the test engineer with AutoCheck (the normal learning curve) and the nature of the test.

A positive side effect is that it is much easier for another test engineer to evaluate the test.

AutoCheck also leads to bigger tests with an increased number of checks for each test case, such as checks for side effects. Such an automatic evaluation tool is also a prerequisite for statistical testing. It would take too much time to evaluate all these test cases manually. In other words, AutoCheck leads to higher test coverage with lower effort for the test engineer.

Once relieved of a great deal of the more mechanical test execution and evaluation activities, the test engineer has time to work on new and better test approaches or possibilities for an increased automation level. Over time this has led to enhancements of both AutoTest and AutoCheck and to new tools like ATP (see **Article 13**).

Acknowledgments

I wish to thank Pascal Lorang for the creation of the prototype and all students involved in the development of AutoCheck.

Reference

1. D. Göring, "An Automated Test Environment for a Medical Patient Monitoring System," *Hewlett-Packard Journal*, Vol. 42, no. 4, October 1991, pp. 49-52.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

MS-DOS is a U.S. registered trademark of Microsoft Corporation.

▶ [Go to Next Article](#)

▶ [Go to Journal Home Page](#)