A High-Level Programming Language for Testing Complex Safety-Critical Systems

Dealing with an enormous amount of data is characteristic of validating complex and safety-critical software systems. ATP, a high-level programming language, supports the validation process. In a patient monitor test environment it has shown its usefulness and power by enabling a dramatic increase in productivity. Its universal character allows it to migrate validation scenarios to different products based on other architectural paradigms.

by Andreas Pirrung

This article concentrates on the specific problem of transforming a test design into concrete automatic test procedures. For a systematic overview and context the reader is referred to ***Article 12***. As described in that article, the test design identifies and documents the test set for a given product. It is derived from external and internal specifications, software quality engineer expertise, and risk and hazard analysis results. A test design is normally informal and describes test cases and test data on a high, abstract level, independent of the test environment. On the other hand, an automatic test procedure has to deal with all the details of the test environment and reflects the abstraction capabilities of the existing tools.

In our software quality engineering department the automatic test environment is based upon two major tools: AutoTest[1] and AutoCheck. The first is a test execution tool and the latter is responsible for test evaluation (see ***Article 14***). AutoTest is very close to the devices it controls and requires detailed commands on a low abstraction level. AutoCheck has to cope with the detailed low-level information produced by AutoTest and therefore also requires input on a detailed, low abstraction level (see Fig. 1). The strengths of the low-level interfaces are their flexibility and adaptability to various different test situations.

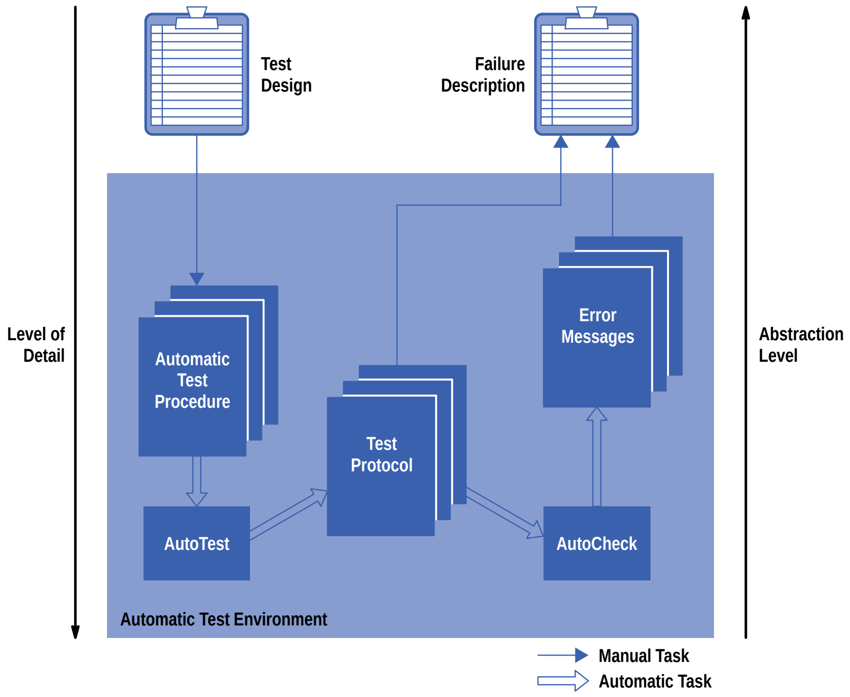

Fig. 1. *Patient monitor test process.*

There are some difficulties with the process shown in Fig. 1. The test engineer spends a lot of time transforming test designs into automatic test procedures. There is a large gap in abstraction level between the test design and the test procedure. The detail level is low in the test design, but very high in the test procedure. It is an error-prone, time-consuming task to bridge this gap manually. Because resources are always restricted, the software quality engineer has less time for a more intensive test design.

Because the test procedures have a high explicit redundancy, it is difficult to maintain and evolve test procedures. The explicit redundancy is high because AutoTest and AutoCheck do not support data and functional abstraction, nor do they offer control flow elements. A piece of code may exist in many copies scattered over the test procedures. If the test requires a change in the code pattern, for instance because of changes in the timing behavior of the system under test (in our case a patient monitor), the test engineer has to update numerous copies of this code pattern. The risk of forgetting one pattern or introducing an error in a test procedure increases with the number of update steps. It is very resource-consuming to adapt test procedures to a change in system behavior.

The test procedure describes a static test scenario. Therefore, the test engineer has to document the test setup completely. Every parameter that influences the test environment and consequently the test execution must be carefully controlled before starting the automatic test. Our test environment consists of so many simulators, forcing devices, and sensing devices that sometimes tests need to be repeated because the initial conditions are wrong. The problem is that the automatic test procedure describes only one specific test situation. It is not possible to use parameters for the test procedure and to feed in the actual start parameters at the beginning of the test execution to get more general and robust test procedures. Even a slight change in the start condition may require an adapted or nearly new test procedure.

The test coverage is limited because the test data is coded within the test procedure. The repetition of a test case with other test data requires a modified duplicate of the test procedure. Again, it would help if a test case were able to profit from data abstraction and parameters, enabling the test engineer to formulate more general test procedures.

AutoTest and AutoCheck do not support the statistical structural testing approach. It is therefore not possible to select test data randomly (see ***Subarticle 13a*** “Structural Testing, Random Testing, and Statistical Structural Testing”).

The following section illustrates the above problems by presenting a practical example to demonstrate the transformation of a high-level test design to an automatic test procedure.

A Practical Example

Patient monitors are electronic medical devices used to monitor physiological parameters of critically ill patients in intensive care units or operating rooms. They alert the medical staff when physiological parameters exceed preconfigured limits. In this example, we will concentrate on a well-known physiological parameter, the heart rate. The nurses and doctors want to get an immediate alarm when the heart rate falls below a given lower limit or exceeds a given higher limit. A malfunction of the monitor may result in the death of a patient, so this functionality is safety-critical and must be validated very carefully by the vendor of the patient monitor. The example illustrates a test design for heart rate alarm testing and the transformation process to the appropriate automatic test procedure.

Fig. 2 shows the upper and lower alarm limits for the heart rate parameter. The data space can be divided into three subdomains (equivalence classes):

- The normal heart rate domain—the interval between the alarm limits. The monitor should not alarm for data points taken from this area.
- The lower alarm domain. All data here produces a low limit alarm.
- The upper alarm domain. All data here produces a high limit alarm.

A classic method of testing the alarm behavior is to select representatives from each of the three areas and check that the monitor reacts as expected. Fig. 2 shows the selected data points and their order in time.

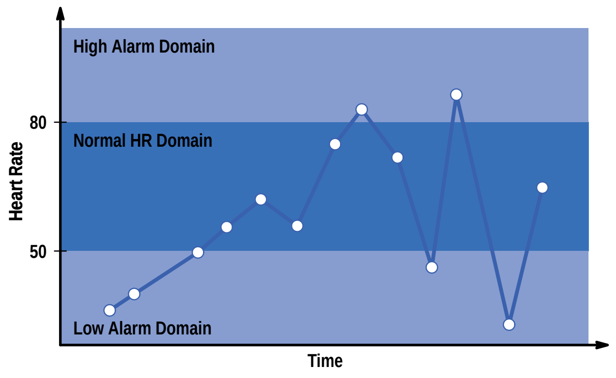

Fig. 2. *Heart rate (HR) alarm test principle.*

This graphical representation of the heart rate (HR) alarm test leads to the following test design:

- Test Case 1:
  - Action(s): Configure HR alarm limits to 50/80. Apply signal HR 45.
  - Expected: Low limit alarm with text “**HR 45 < 50”.
- Test Case 2:
  - Action(s): Apply HR signal 49.
  - Expected: Low limit alarm remains with text “**HR 49 < 50”.
- Test Case 3:
  - Action(s): Apply HR signal 50.
  - Expected: Low limit alarm disappears.
- Test Case 4:
  - Action(s): Select 5 different HR values between 50 and 80 and apply them.
  - Expected: No HR alarm for each of the selected values.

These few test cases are enough to demonstrate the principles of test design. The appropriate automatic test procedure description for test case 1 on the AutoTest and AutoCheck level then looks like:

```
.....
//--------------- Test Case 1: -----------------
// Adjust alarm limits to 50-80.
//
merlin param
merlin ”HR”
merlin f2
merlin f7
merlin f3 -n48
merlin f6 -n12
merlin f4 -n34
merlin f5 -n5
//
// Apply HR signal 45 (normal sinus beat).
//
sim1 NSB45
//
// Delay 10 s : after that time the alarm must
// be announced.
wait 10
//
// Check if low limit alarm ”** HR 45 < 50” is
// present.
^ Verify begin
^ Alarm ”HR” < al_min;
// Low alarm active.
^ sound is c_yellow;
// Limit alarm sound audible.
^ Verify end
mecif tune HR 10
// Tune 10 the HR numeric of the patient
// monitor.
.....
```

This example demonstrates the difference in abstraction between a test design (high abstraction) and an automatic test procedure (low abstraction) and gives an impression of the difficulties noted above. An automatic test description language designed to alleviate these difficulties should offer abstraction capabilities to hide details and to compose complex functions from simpler functions. Like every high-level programming language, it should bridge the abstraction gap automatically. In the following section a solution is presented that meets these needs.

The ATP Programming Language

Often specific problems need *basic processors* like AutoTest or AutoCheck to perform some operation such as pushing keys, simulating patient signals, simulating powerfail conditions, and so on. A straightforward solution might extend the command interfaces of AutoTest and AutoCheck to support data and functional abstraction, provide control flow elements like conditions and loops, and allow further probabilistic data generation. This would probably eliminate the difficulties mentioned above. However, redundant effort would have to be spent implementing an abstract command interface again and again.

Our solution is ATP (Automated Test Procedure), a high-level programming language that offers the abstraction facilities and makes it possible to integrate basic processors smoothly. ATP allows the integration of many different basic processors, so the coordination of the basic processors is much easier than with separate control.

The following is a typical ATP routine representing the automatic test procedure for the heart rate alarm test:

```
DEFINE AlarmTest ( IN PatientSize CHECK IN {
              ”ADULT”, ”PEDIATRIC”, ”NEONATE”
                  },
              IN Category CHECK IN {”OR”, ”ICU”}
                 )

DESCRIPTION
  PURPOSE :
    This routine demonstrates some of the
    ATP features. It is an automatic test
    procedure testing the HR alarm
    capabilities.

  SIGNATURE :
    AlarmTest ( <PatientSize>, <Category> )
END DESCRIPTION

LOCAL   HRValue,/* selected HR Value          */
        AL,     /* HR low alarm limit         */
        AH,     /* HR high alarm limit        */
        walk    /* repetition counter         */

/*–––––––– Initialize the Repository –––––––––*/

LINK Repository <– ”$PatientMonitorRepository”
Repository:Init (PatientSize, Category)
              /* Declare the use of a function
                 repository and initialize the
                 repository link. This gives
                 context–specific access to all
                 available functions for the
                 given PatientSize and Category.
                                              */

/*––––––––––––––– Test Case 1 ––––––––––––––––––
  Action(s): Configure HR Alarm Limits to 50/90.
             Apply Signal HR 45.
  Expected:  Low limit alarm with text
             ”**HR 45<50”.
  ––––––––––––––––––––––––––––––––––––––––––––––*/
HR:SetAlarmLimits (50, 90)
HR:SimulateValue (45)
HR:CheckAlarm (10, ”** HR 45 < 50”)
              /* Check for limit alarm after
                 delay of 10 s.               */

/*––––––––––––––– Test Case 2 –––––––––––––––––
  Action(s):  Apply Signal HR 49
  Expected:   Low limit alarm remains with text
              ”**HR 49<50” (alarm string is
              updated without delay).
––––––––––––––––––––––––––––––––––––––––––––––*/
HR:SimulateValue (49)
HR:CheckAlarm (0, ”** HR 49 < 50”)
```

```
/*--------------- Test Case 3 -----------------
  Action(s):  Apply Signal HR 50.
  Expected:   Low alarm limit disappears.
----------------------------------------------*/
HR:SimulateValue (50)
HR:CheckNoAlarm (5)
            /* No HR alarm present after 5 s. */

/*--------------- Test Case 4 -----------------
  Action(s):  Select 5 different HR values
              between 50 and 80 and apply them.
  Expected:   No HR Alarm for each of the
              selected values.
 ---------------------------------------------*/

walk <– 1
              /* Randomly choose some HR values
                 in the range 50/80, i.e., no
                 alarm condition exists and
                 therefore no HR alarm must
                 be visible and audible.      */
WHILE walk <= 5 DO
  HRValue <– RANDOM (50, 80, 1)
  HR:SimulateValue (HRValue)
  HR:CheckNoAlarm (0)
  walk  <– walk + 1
ENDWHILE

.....

END AlarmTest
```

An automatic test procedure for a random heart rate alarm test in ATP might look like the following:

```
DEFINE RandomAlarmTest ( IN repetitions )

DESCRIPTION
  PURPOSE :
    Random HR Alarm Test

  SIGNATURE :
    RandomAlarmTest ( <repetitions> )
END DESCRIPTION

LOCAL   AL,     /* low alarm limit            */
        AH,     /* high alarm limit           */
        walk,
        HRValue

/*-------- Initialize the Repository ---------*/

LINK Repository <– ”$PatientMonitorRepository”
Repository:Init (CHOOSE ({”ADULT”,”PEDIATRIC”,
                         ”NEONATE”}),
                CHOOSE ({”OR”, ”ICU”})
                )
               /* Declare the use of a function
                  repository and initialize the
                  repository link. Choose
                  patient size and category
                  randomly. This gives context-
                  specific access to all avail-
                  able functions for the given
                  PatientSize and Category.   */


/*--------------------------------------------*/
                /* Randomly select valid HR
                   alarm limits, then randomly
                   select an HR value and check
                   if the monitor reacts as
                   expected.                  */
walk <– 1
WHILE walk <= repetitions DO
  HR:RandomSelectAlarmLimits (AL, AH)
                /* randomly select valid alarm
                   limits                     */
  HR:SetAlarmLimits (AL, AH)
  HRValue <– RANDOM (20, 180, 1)
                /* select HR values between 20
                   and 180 with step width 1. */
  HR:SimulateValue (HRValue)
  IF HRValue < AL THEN
    HR:CheckAlarm (10, ”** HR ” + HRValue + ”<”
                   + AL)
  ELSIF HRValue > AH THEN
    HR:CheckAlarm (10, ”** HR ” + HRValue + ”>”
                   + AH)
  ELSE
    HR:CheckNoAlarm (5)
  ENDIF
  walk    <– walk + 1
ENDWHILE

END RandomAlarmTest
```

Even without familiarity with the syntax and semantics of the ATP language, it can be recognized that the abstraction level is higher than with plain code for the basic processors (in our case, AutoTest and AutoCheck). It is also worth noting that the automatic test procedures are not restricted to a specific patient monitor. They describe in a general and abstract way a heart rate alarm test for any patient monitor with a limit alarm concept. The differences between specific patient monitors are in the basic processor interfaces and in the primitive functions.

The ATP Concept

ATP consists of two major functional elements (see Fig. 3): a set of tools to maintain a function repository and an ATP language interpreter.

The tools give the user adequate access to the function repository, which contains well-documented, well-tested ATP functions that can be reused. This effectively reduces redundancy and increases productivity (see the next section, “Working with the Function Repository”). The tools can be grouped into:

- File-oriented tools. Check functions into and out of the function repository, compare two versions of a function, etc.
- Repository query functions. Obtain information about available functions, about an interface of a function, etc.
- Repository administration functions. Administer and maintain the structure of the repository. Allow archival and retrieval of the repository. These functions are only accessible by the repository administrator.

With these tools and a text editor, a programmer writes ATP functions by reusing existing functions from the repository. When the function repository is well-structured and offers reliable functions on an adequate abstraction level, it is easy even for an inexperienced programmer to write functions, as shown in the example above.

Interpreter

The core of the system is the ATP language interpreter. The interpreter requires an input file and an output file. The input file is an ATP function. In contrast to other common high-level languages, ATP requires one file per function. One advantage of this approach is that each ATP function is executable. There is no explicit syntactical distinction between a main routine and a subroutine. Any function can be the execution starting point and can call any other function. There is no explicit function hierarchy. The hierarchical structure is provided independently by the repository structure.

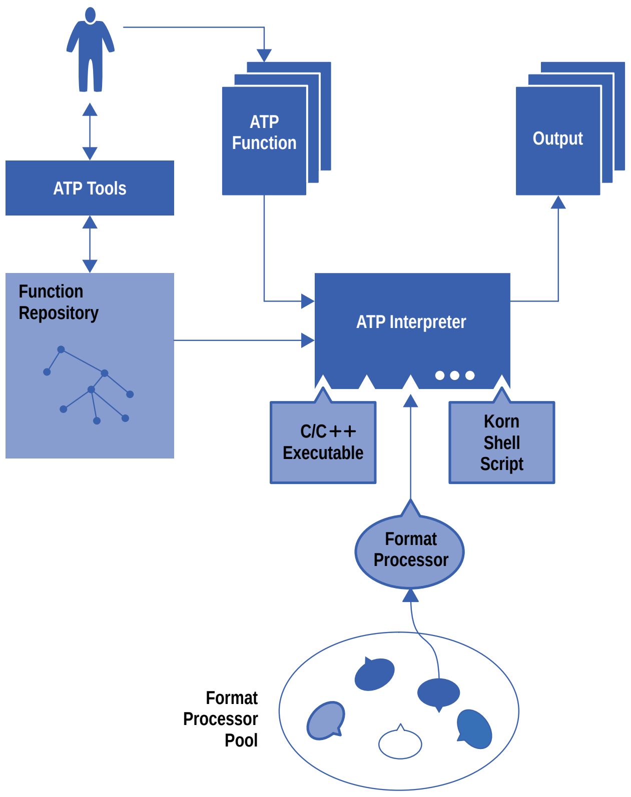

Fig. 3. *ATP concept overview.*

Another advantage is that, because each file contains only one function, it is much easier to administer the functions in the function repository. To fulfill structural requirements the functions can be grouped by any criteria. The heart rate example above uses functions that all belong to the logical functional group HR.

The interpreter output data can be written into an output file. A powerful feature of the language is its ability to integrate *format processors*. A format processor is a problem-domain-specific process (a basic processor) that can be integrated into the interpretation process. In the patient monitor testing example, AutoTest and AutoCheck are format processors. The ATP language offers syntactical elements to establish a communications channel to a format processor so that within the ATP code the user can send any information to the processor. The format processor can send back information to the ATP interpreter, which then can be sent to another format processor or logged to the output file. The creation of this ATP adapter interface is an easy task, thanks to an API that enables a programmer to implement this communication interface with ATP. If an integrated format processor is general-purpose, it can be offered to all ATP programmers. A good example is KSH, a format processor that enables ATP programmers to integrate Korn shell commands within ATP code. The format processor concept and the abstraction facilities of the ATP language offer the programmer the means to model the problem domain in an adequate and flexible way.

Working with the Function Repository

The following short tour illustrates some of the tools that are available to handle function repositories. Suppose that a programmer wants to know which functions in the repository are available for dealing with heart rate operations. Typing:

```
LibIndex
Group: ”HR”
```

will, for example, produce the output:

```
CheckAlarm
CheckNoAlarm
SimulateValue
SetAlarmLimits
RandomSelectAlarmLimits
.....
```

To get information about the interface of the SetAlarmLimits function, the programmer can type:

```
TellMe
Group: ”HR”
Functions: ”SetAlarmLimits”
```

and will get:

```
************** HR:SetAlarmLimits **************
*
* purpose: configure the Heart Rate lower and
* upper alarm limits.
*
* signature: SetAlarmLimits(<lower alarm limit>,
*                   <higher alarm limit>)
*
* .....
```

The heart rate test example above uses this and other functions. At the beginning of the function a LINK statement declares an access path to a given function repository. Then the function repository is initialized. This initialization function introduces all available functional groups given a specific system context. At that point the programmer is able to call the functions, for example HR:SetAlarmLimits, without knowing implementation details or physical locations. It is possible to check out a function for enhancement or maintenance purposes. It is also possible to check a function into the repository so that all test engineers can use the new function.

Format Processors

Fig. 4 presents the possibilities and the flexibility of format processors. Each format processor consists of two parts: a basic processor and an ATP adapter. The basic processor is a proprietary part, that is, any executable code written by a programmer. Typically the basic processors are on a low abstraction level. The ATP adapter is the interface to ATP that allows data to be sent to ATP and received from it. This functionality is encapsulated and offered as an API.

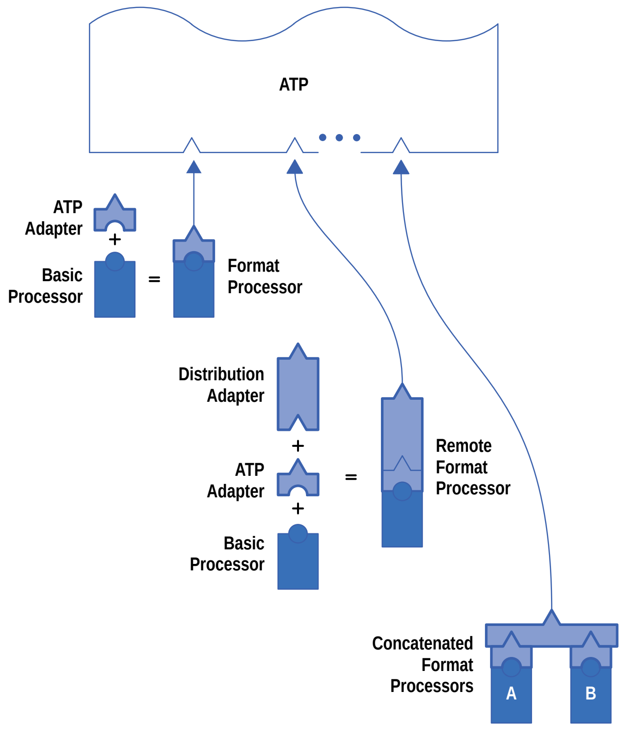

Fig. 4. *Format processor functional blocks.*

A format processor can be used within ATP in the following way:

```
.....
FORMAT MyFormatProcessor <- ”$MY_FP_EXECUTABLE”
/* Declare the use of a format processor. */
.....
BEGIN [ MyFormatProcessor ]
.....
.....
END [ MyFormatProcessor ]
/* Use the format processor, sending and receiving information. */
.....
```

First, a specific syntactical construct introduced with the keyword FORMAT is used to declare the use of a format processor. It is then the task of ATP to control and to communicate with the format processor. All information enclosed in the syntactical bracket BEGIN [MyFormatProcessor] and END [MyFormatProcessor] represents a code template for the named format processor. ATP generates the actual code block from this code template by substituting the actual parameter values for the code template parameters. Then this code block is sent to the format processor for immediate execution. The format processor receives the code by calling API functions provided by the ATP adapter. The proprietary part of the format processor processes the received information. The format processor can send back information to ATP. ATP receives this information and logs it to the output file or redirects it to another format processor.

Remote Format Processor. If a programmer needs to integrate a format processor on another machine, for example on a PC running Windows® NT, this can be specified in the FORMAT declaration. No additional effort is required for the programmer to establish a remote format processor. The adaptation for remote control is done by ATP automatically by adding a distribution adapter.

Concatenated Format Processor. Another feature of the ATP language is the ability to concatenate existing format processors.

```
.....
FORMAT X <- .....
FORMAT Y <- .....
FORMAT Z <- X | Y
/* Concatenate format processors X and Y to Z.
This is similar to UNIX pipes. */
.....
BEGIN [ Z ]
.....
END [ Z ]
```

The code block between BEGIN [Z] and END [Z] is first sent to format processor X. The output of format processor X is sent to format processor Y. For format processor Y it makes no difference where the information comes from, that is, the concatenation is mediated by ATP automatically. Format processor Y sends its output back to ATP.

ATP in the Patient Monitor Test Environment

The concept behind ATP eased its integration into the patient monitor test environment. The impetus to develop this concept came from our experience with the test environment, as described at the beginning of this article. But the concept is more general. It is not resticted to the patient monitor test environment. ATP can be used to attack many different problems.

The integration of such a tool into an existing environment is a challenging task. ATP, like every tool, requires some effort to build up the necessary infrastructure, to support the tool, and to learn the new language. A step-by-step, three-phase integration of ATP in the test process was planned.

Phase I. Develop a patient-monitor-relevant repository structure that is easy to use and maintain. In parallel, implement a set of primitive functions to fill the repository. Then test the structure on new patient monitor functionality. In this phase ATP does not specify any format processor executable, that is, AutoTest and AutoCheck are not integrated as format processors. ATP writes the code block immediately to the output file. The generated code is then processed in a postprocessing step.

Phase II. Enhance AutoTest and AutoCheck with ATP adapters so that they can be integrated as format processors. Then the same functions used in phase I can be executed immediately without the postprocessing steps needed in phase I. The test tools are invoked by ATP.

Phase III. Complete the function repository and migrate, step by step, the existing test package to ATP functions. Then the testing package can be used again for new patient monitor products simply by replacing the format processors by new ones and by substituting some primitive functions.

Currently the phase I integration is completed (see Fig. 5). A repository structure has been proposed and evaluated in some projects. In these projects test engineers use ATP to automate the tests. ATP generates AutoTest and AutoCheck code, which is then passed to AutoTest and AutoCheck for execution. For phase II integration, only the declaration of the AutoTest and AutoCheck format processors will change. They will then specify integratable AutoTest and AutoCheck format processors. This phase is currently in progress. Phase III has been started.

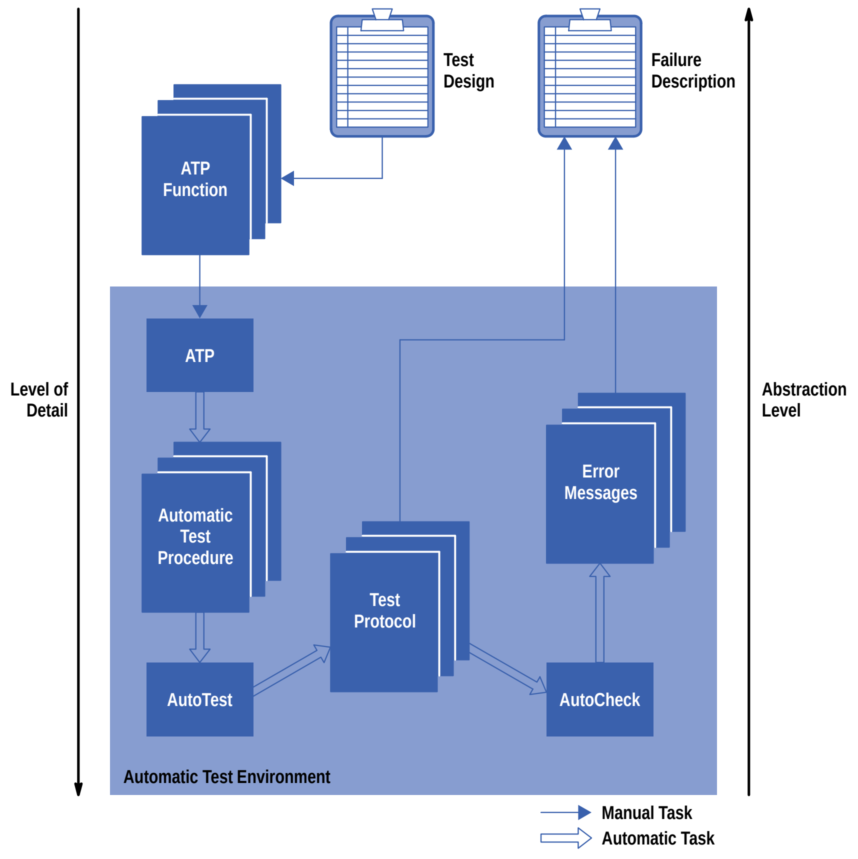

Fig. 5. *Current ATP integration in the patient monitor test environment (phase I).*

ATP Integration in Phase I: An Example

The following ATP function illustrates how ATP generates AutoCheck code. This function is the CheckAlarm function called in the heart rate alarm test used in the example presented earlier.

```
DEFINE CheckAlarm ( IN AlarmDelay TYPE IN
                      {"REAL"},
                    IN AlarmString TYPE IN
                      {"STRING"}
                   )

DESCRIPTION
  PURPOSE :
    Check if alarm is present after a specified
    delay.

  SIGNATURE :
    CheckAlarm ( <AlarmDelay>, <AlarmString> )
END DESCRIPTION

FORMAT  AutoTest  <- " "
FORMAT  AutoCheck <- " "
       /* At the moment AutoTest and AutoCheck
          are not really format processors. The
          declaration of AutoTest and AutoCheck
          does not specify any format processor
          executable. In this case ATP writes
          the code block immediatly to the
          output channel, i.e. ATP generates
          AutoTest/AutoCheck code.            */

LOCAL sound

IF ”***” == AlarmString [1, 3] THEN
       /* Is it a red alarm?                  */
  sound <– ”red”
ELSE
       /* NO ==> yellow alarm                 */
  sound <– ”c_yellow”
ENDIF
       /* Very critical alarms are announced as
          red alarms whereas less critical
          alarms are announced as yellow alarms.
          Parallel to the visible colored alarm
          string a corresponding sound is
          audible, i.e. for red alarms a red
          alarm sound is audible and for yellow
          alarms a c_yellow alarm sound is
          audible.                           */

BEGIN [ AutoCheck ]
  ^ Verify begin
  ^    Alarm @(1,AlarmString) within
             (@(1,AlarmDelay),NaN);
  ^    sound is @(1,sound);
  ^ Verify end
END [ AutoCheck ]
       /* AutoCheck code generation.
          The code template includes ATP
          variables, which will be evaluated
          at run time.                       */

END CheckAlarm
```

Discussion

Although the current ATP integration is only the first phase, ATP has proved to be a powerful tool for attacking and solving complex testing problems that otherwise would not have been solved in the same time frame. Like every new tool, at the beginning some effort is required to learn the language. Also, the test engineers have to implement a set of primitive functions to build a powerful function repository. Nevertheless, our experience has shown that productivity increased significantly and that ATP helped to ensure the predictability of product releases.

After a few days of use the test engineers felt comfortable enough to develop their first automatic tests with ATP and were able to use the function repository.

Tests are much more sophisticated and effective than before. The same tests written directly in Autotest/AutoCheck code would have probably required three times more development time without reaching the same degree of reliability, flexibility, and maintainability. The test engineers using ATP used the increased productivity to think about better test designs.

Failures have been found earlier because of higher test coverage, especially from the use of random test data generation. These failures would not have been detected in the validation phase with the existing static test. The risk of missing a failure is therefore reduced by ATP.

The redundancy of the tests is much lower. Test engineers are now able to adapt their test procedures rapidly to changed system behavior. In most cases they just have to update some constants.

The higher abstraction level of the test procedures enables the test engineer to use the same test procedures to test new patient monitor products. The adaptation requires the substitution of some low-level primitive functions and the format processors.

Implementation

The ATP interpreter is implemented in C on a workstation running the HP-UX* operating system. Most of the repository tools have been written in the ATP language itself. This illustrates that ATP is not only a language for formulating test procedures.

The architecture follows the classical compiler architecture. The front end with lexical analysis, syntactical analysis, and semantic analysis is similar to other compilers for high-level formal languages. The back end consists of the code generation module and the communication module, which manages the format processor communication and other functions.

Conclusion

The new ATP language bridges the gap between high-level test design and low-level automatic test procedures. The integration of ATP into the test environment has increased productivity and reduced redundancy. More important, the quality of the testing process has increased with the use of this abstract high-level programming language. Migration of the test procedure set to new products is now much easier because most of the code can be reused.

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.

▶ Go to Subarticle 13a

▶ Go to Next Article

▶ Go to Journal Home Page