ISP RAS

CTesK

Automating Testing of C Applications

The problem of testing

Nowadays, testing costs may exceed one half of the whole budget in a typical software development project. Moreover, it is not unusual, when the testing efforts during the product maintenance after its release cost more than the product development. At the same time the effectiveness of testing is not excellent. It leads to objectionable economics impacts and, sometimes, to accidents with inadmissible consequences.

The main reasons are the growth of software size and complexity and the cutting down of time-to-market. To conform the challenges of real world software testing needs more systematic and flexible approaches, more automating and reuse.

CTesK provides such approach and cost-efficient way of automating to an important kind of testing — *functional testing*. Also it is called *conformance testing*, more details of this concept are explained in next section. The product allows giving *more* tests of *higher* quality with *less* money. Many people associate with a concept of automatic testing some framework of executing hand-written scripts. CTesK is not another such framework. Rather, it is a framework for derivation and execution of test cases.

Functional testing

There are many kinds of testing: performance testing, stress testing and so on. But whether an implementation meets performance and resource requirements or not, its behavior should be proper. In other words the system behavior should *conform* to *functional requirements*.

Any software communicates with the external world via some interface. For example, it may be API of the software component. Functional requirements do not describe how the system should be implemented. They define what *externally observable* effects the system must produce when interacting with the environment by means of *an interface* of the system.

System behavior conforms to its functional requirements if any effect that is being observing is judged by the functional requirements. Thus the goal of functional testing is verifying that an implementation does that it has to do and does not that it has not to do. Functional testing should cover systematic errors with respect to violations of functional requirements. These violations typically result from misunderstanding between customers and developers, between designers

Copyright @ 2003 Institute for System Programming of Russian Academy of Sciences.
All rights reserved

ISP RAS

and implementers, from requirements changes during the projects. Detecting such errors as quickly as possible is extremely critical to reduce risks of failing system integration.

Many software testing efforts can be called either functional testing, error hunting, stress testing or somehow else. No matter what it is called functional testing is a major part of software testing efforts.

CTesK approach

Automating functional testing is possible only if functional requirements are specified in a strong formal way. Formal means: according to *a computer readable* form that has *a unique interpretation*. It is not bound to difficult mathematics or theoretical considerations. The difference between informal and formal specifications of functional requirements is not between programming and mathematics languages, rather it is between natural and programming languages.

CTesK works with assertions about system behavior in the forms of pre- and postconditions. They are defined in specification extension of C — SeC. SeC is a compact and enough comprehensible language to experienced C developers.

The assertion based formal specifications define *a state-based model of the system behavior*. They also contain a description of coverage criteria of requirements. CTesK test environment provides the automatic monitoring of the percent of achieved coverage. CTesK generator is used to build components in C, which check correctness of the behavior of an implementation under test with respect to the specified model.

To build automatically a *test sequence* (a sequence of various test actions via an interface of the system) mathematical algorithms implemented by *test engine* are used. These algorithms ensure exploring a vast number of states and checking the system behavior in each of ones with required coverage. Test case developer should provide a short description of test case: what parts of the system interface should be tested, how parameters of these interface parts should be iterated and what state should be used in the test (the test state encapsulates a history of system-environment interaction during test execution). Test case descriptions are developed in SeC in *test scenarios*, which are sheets of test cases.

To enable an integration of test cases into the implementation under test *mediators* in SeC are developed. They are adapter sheets binding the specification model and the implementation. CTesK generator translates them into mediators in C.

Adopting CTesK approach into software development process

Usually software development begins with elicitation and analysis of stakeholder needs. These are transformed into architectural description and software requirements of smaller and smaller components. Components are implemented based on their software requirements. Then they and

Copyright @ 2003 Institute for System Programming of Russian Academy of Sciences.
All rights reserved

their interactions are tested during unit and integration testing to verify the correspondence to their software requirements.

To be able to automate testing activities using CTesK, software requirements should be formal in the sense explained above. Although formal specifications is introduced by CTesK to automate testing, they are also useful in others activities of software development. By virtue of their rigor, formal specifications require a developer to think out his design in a more thorough fashion. Unique interpretation of formal specifications removes ambiguities and inconsistencies in software requirements. It decreases misunderstanding between the developers. Therefore, creating formal specifications can help to identify errors far earlier than in traditional design.

Thus the best way of using CTesK approach in a software development is not to introduce separate activity of testing with CTesK, but rather using of formal specifications in early development stages and an automated development of integration and unit tests in concurrent with the design and implementation development. The figure below outlines such approach.

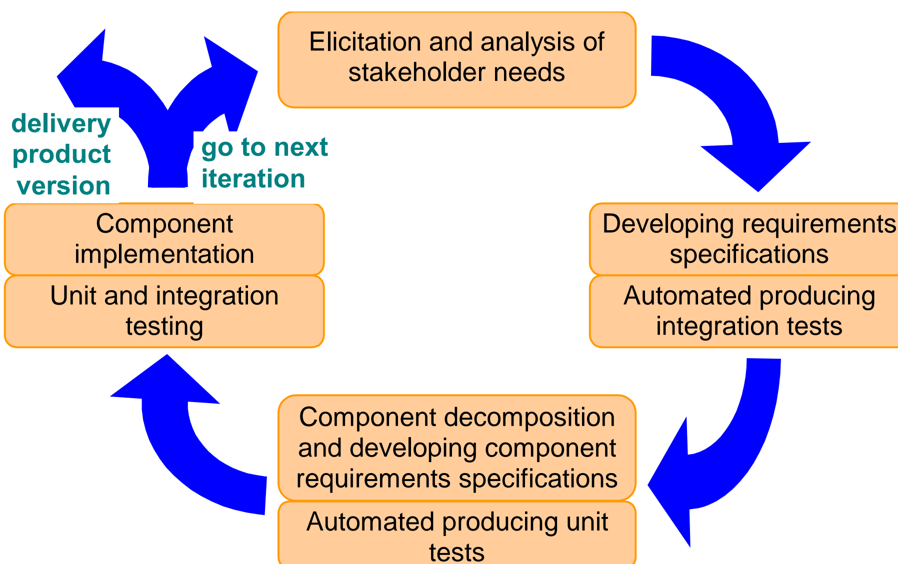

Figure 1 Introducing CTesK approach into software development

However, an introducing formal specification into software development process is often very difficult due to such factors as customs and traditions, policies and problems of lacks of time and resources required to training effort. In such cases there are various ways of an adopting CTesK approach in a less scales. CTesK can be used for automated unit testing of one or few components, or it can be used for only automated integration testing. Also CTesK is applicable in reverse engineering projects. Besides, formal specifications can define a high level model of system behavior, which is independent of implementation details. By means of mediators CTesK offers a flexible integration of generated tests into different implementations. Thus, CTesK provides a powerful support of automated development of regression tests and automatic regression testing.

Copyright @ 2003 Institute for System Programming of Russian Academy of Sciences.
All rights reserved

Previously discussed ways of adopting CTesK can be applied in initial pilot projects. Later if the organization sees that the formal specifications actually work and provide additional value, their use can be expanded from the testing area to other areas as well.

The main features and testing architecture

CTesK is a framework for automated functional testing based on formal specifications. Its main features are the following:

- Requirements specification by using SeC — compact and comprehensible specification extension of C programming language, defining high level models of system behavior from which test oracles are automatically generated.
- Specification-based coverage metrics, quantifying the extent of testing and offering meaningful test completion criteria.
- Test case generation from test scenario — short sheet of test case in SeC.
- Automatic generation of systematically relevant test sequence during test execution.
- Flexible test integration into an implementation under test by means of mediator generation from SeC mediator sheets.
- Support for testing distributed systems and systems with asynchronous interfaces.
- Automatic test execution.
- Generation error and coverage reports from the test data.

Overall architecture of an automated testing with CTesK can be illustrated by the following figure.

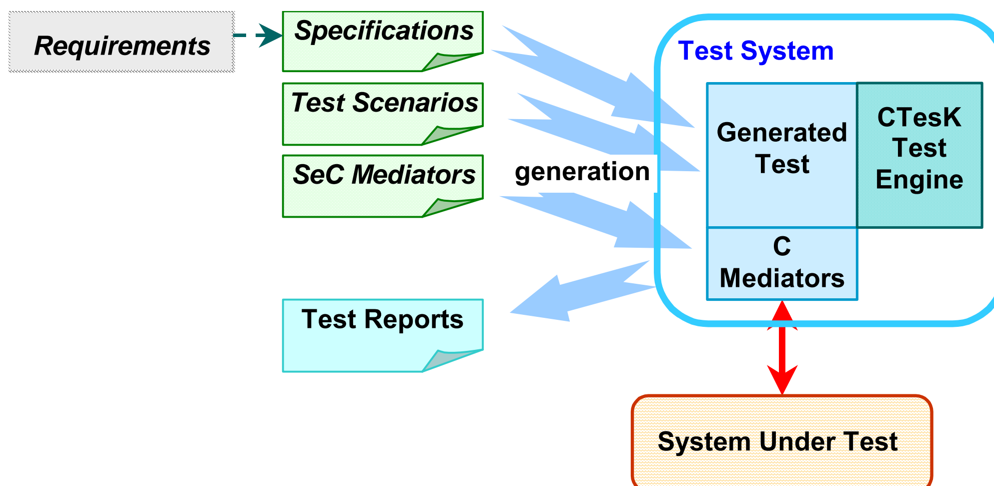

Figure 2 Overall architecture of CTesK automated testing

The following process of an automated functional testing can be implemented using CTesK:

1. **Decision on area and depth of testing.**
   At this step, the set of target system components to be tested with CTesK are determined.

Copyright @ 2003 Institute for System Programming of Russian Academy of Sciences. All rights reserved

For each component, its interface is designated and basic testing requirements, which define a level of abstraction for model of this component, are determined.

2. **Development of behavior model to be tested.**
   At this step, formal specifications of behavior for each component to be tested are developed.
3. **Structuring of test suite.**
   At this step, the goal of testing and its representation as the test coverage level for specifications and source code of the selected components are defined. It is called *the target coverage level*. Then, a number of test cases to achieve the target coverage level are designed. These test cases can be of different kinds; they may test different things, and may be differently constructed. The components of the target system and their interfaces touched by the test for each test case in the suite are determined.
4. **Mediator development.**
   At this step, SeC mediators for each component to be tested are developed. A mediator binds the model with the implementation under test, the functionality of which is described in this model. One mediator can be used in several test cases.
5. **Test scenario development.**
   At this step, test scenario for each designed test case is prepared. Test scenario describes the generalized state of the set of target components and a number of test actions to be performed on these components in each state. Actually, test scenario implicitly describes an FSM model to be used for automatic test sequence generation.
6. **Test suite debugging.**
   At this step, all developed test components are integrated into the implementation under test and debugged. All the errors in the components of the test system (not in the target one!) should be fixed before the next step can be performed.
7. **Test execution and test results analysis.**
   At this step, the debugged tests are executed. The resulting reports are analyzed to detect errors in the target system and to determine achieved coverage level.

After running some tests the construction of new complementary test cases based on coverage information can be required.

The testing process is called "*automated*" because it automates

- generation of test components from specifications and test scenarios
- generation of test sequences
- execution of tests
- generation of error and coverage reports

The application areas

There are no fundamental limitations on the kind of software under test. CTesK is very suitable for testing any software components with Application Program Interface. In other cases such as testing of distributed systems and systems with asynchronous interfaces (for example, based on events or messaging) special features of SeC can be used to describe the behavior of the system

Copyright @ 2003 Institute for System Programming of Russian Academy of Sciences.
All rights reserved

under test and to construct relevant test scenario. The test engine automatically checks the correctness of the system behavior when asynchronous and concurrent interactions between the system and its environment occur.

Nevertheless, tools like CTesK are usually applied to software with high quality (reliability) requirements. So, the most perspective areas of CTesK application are as follows:

- Mission-critical software (control systems in airspace and defense applications, embedded software, industrial process control, health monitoring systems).
- System software like components implementing OS services, Web servers.
- Telecommunication software.
- Any software related to formally stated standards (like protocols, languages, and Internet applications).

The benefits

Most valuable benefits of the CTesK deployment are as follows:

- **Automation of test production**
  CTesK generator provides capabilities to generate test component from requirements specifications and test scenarios, which are developed in SeC — compact and comprehensible specification extension of C programming language. Thus, user should not master some new specification language.
- **Automatic verdict assignment.**
  CTesK test **automatically assigns a verdict** on the system conformity with its specifications. The verdict is based on the results of system behavior checking performed during testing.
- **Automatic test sequence generation**
  With the help of relatively small and simple test scenarios and specifications of the target software CTesK can automatically generate effective tests based on mathematical algorithms. Such tests check the behavior of the target software in all substantially different its states defined by specifications. This feature completely eliminates the need of test scripts making up the most part of a traditional test suite.
- **Testing of asynchronous and concurrent interfaces**
  SeC provides special features to specify the systems with asynchronous interfaces, for example based on events or messaging, and to construct test scenario with concurrent interactions (synchronous and asynchronous) between the system and its environment. The test engine automatically checks the correctness of the system behavior when asynchronous and concurrent interactions occur.
- **Improving quality of testing**
  Test sequences generated by CTesK test engine explore a vast number of states. They are very systematically relevant and ensure required test coverage.
- **Regression testing**
  CTesK provides full support for regression testing. The flexible mechanism of binding specification and implementation allows checking next version of some software component against any of its old specifications.

Copyright @ 2003 Institute for System Programming of Russian Academy of Sciences.
All rights reserved

- **High reuse of specification and handmade components of test suites**
  By virtue of a high level of models defined by specifications and an independence of specifications from the implementation, a high level of reuse of specifications and other hand made components of test suites is possible.
- **General improvement of software development process**
  Besides improvement of test production process itself, CTesK approach can help to improve general software development process by forward development of tests concurrent with the development of the system. It leads to reducing duration of software development cycle.

Technical description

Current version of CTesK runs on any of the following operating system platforms:

- Windows(2000/XP)
- Linux (Red Hat 7.0 or newer)
- Other operating systems under separate contracts

CTesK also requires JDK 1.4.1, Microsoft Visual C (Windows), gcc (Linux).

CTesK version integrated into MS Visual Studio 6.0 requires MS Visual Studio Service Pack.

Copyright @ 2003 Institute for System Programming of Russian Academy of Sciences.
All rights reserved