

Testing Safety-Critical Systems - A Reuse-Oriented Approach

Mustafa Poonawala[†], Satish Subramanian[†], Wei-Tek Tsai[†],
Ramin Mojdehbakhsh[‡], Lynn Elliott[‡]

[†]Department of Computer Science,
4-192 EE/CS Building, 200 Union St.,
University of Minnesota, Minneapolis, MN 55455.
{mustafa, subraman, ramav, tsai}@cs.umn.edu

[‡]Guidant Corporation
Cardiac Pacemakers Inc.
St. Paul, Minnesota.
{mojdehbakhsh, elliot}@lilly.com

Corresponding Author:

W.T. Tsai

Department of Computer Science,
4-192, EE-CS Building,
200 Union Street,
University of Minnesota,
Minneapolis, MN 55455, USA.

email: tsai@cs.umn.edu

Phone: 612-625-6371

Fax: 612-625-0572

Abstract

This paper discusses the testing of a safety-critical medical device in an industrial environment. The authors have worked on the development and testing of a cardiac rhythm management system at Guidant Corporation, which is involved in the development of a family of related medical devices. The testing process of these systems is expensive because of the stringent safety and reliability requirement of these devices. To leverage the cost involved in the testing process we take advantage of the overlap in functionality across a family of products. In this paper, we present a domain-specific reuse approach we used in testing these safety-critical software systems. The approach allows easy generation of test artifacts, like test scenarios and test cases, while maximizing reusability. We have demonstrated our technique in the testing of a cardiac pacemaker and have achieved significant improvements in productivity.

1. Introduction

This paper discusses testing of a safety-critical system in an industrial environment. Safety-critical systems, which consists of a variety of software, hardware and firmware components, need to be rigorously tested for safety and reliability. Identification and removal of faults from these systems is of utmost importance due to the safety critical nature of the products [FDA 89, FDA 91, Mojdehbakhsh 94a]. Guidant Corporation has been involved in the development of

implantable medical devices, such as cardiac rhythm management systems, which are complex real-time safety-critical systems. These devices have approximately hundreds of thousand lines of code and required to be reliable, safe and efficient to control both the external and internal operations of the cardiac pacemaker systems [Elliott 94, Mojdehbakhsh 94b]. The authors have worked on the development of a cardiac rhythm management system and were involved in the testing of the system. Here, we discuss our experience in using a systematic reuse-oriented approach in testing these safety critical medical devices.

1.1. System Characteristics and Implications

Safety-critical systems that we dealt with in an industrial environment have the following characteristics.

1. The products were mission-critical real-time systems and so they undergo a rigorous multiple stage of testing and revalidation, which include requirement validation, design and code reviews, unit, component and integration testing, and field evaluation.
2. Products are constantly evolving as technology and market needs change.
3. The systems being developed are a family of products, where the products are evolving from one another, and share a lot of functionality among them.
4. Engineers usually work against pre-determined schedules and have specific deliverables assigned to them. So there is concurrent development and testing of the products at different stages at a given time.
5. Experts from different domains work on the development of a product at various stages. These include physicians, software engineers, electrical engineers, industrial and manufacturing engineers.

As we are dealing with testing, we discuss the implications of these characteristics on the testing process.

- a) *Simple and effective*: Due to the complexity of the system, and the various safety and reliability requirements, testing is critical and requires significant effort. Often a test engineer requires several weeks to develop and execute a test case for a specific task of the system due to the complexity of the system. Any technique that can make this testing process efficient and effective is highly desirable. In addition, the tight schedules and pre-defined tasks makes it difficult for the engineers to switch to newer technology that have steep learning curves. Thus, the technique must be simple, easy to understand, and can be learned in a short period.

These requirements are dictated by the system characteristics described in items 1, 2 and 4 above.

- b) *Reusability*: As products evolve into a family of products, the test cases developed for these products by various test engineers have significant overlap among them. Due to the complexity involved in the system, reuse of these test cases within and across products has been difficult. The time and effort in this step can be drastically reduced if commonality among products is reused. This follows from items 2 and 3 above.
- c) *Abstractions*: In a typical industrial environment, different engineers are involved in product development at various stages (item 5 above). The domain experts or system analysts develop the system requirements. The software engineers design and implement the software for the requirements. So the software engineer must have some knowledge of the domain to understand the requirements, and the domain experts must have some system knowledge to shape the requirements. But typically, domain experts do not know everything about the software architecture and test environments to test if the software system satisfies the requirements. A software engineer does not know the domain enough to see if the software satisfied the domain-level requirements. This lack of respective domain knowledge often lead to long testing cycles. It is desirable that the domain experts and the software engineers have a common domain vocabulary that both can understand and use. The domain expert can use the vocabulary to specify the functionality of the system and software engineer can use it to develop test cases.

This paper proposes an approach that assists test engineers, developing a family of products, in improving the productivity of the testing process. This approach takes advantage of the potential for reusability that exists in the system. This process emphasizes the reuse of existing test artifacts from earlier testing stages (within product) and from earlier product testing (across products).

Section 2 describes the issues and problems in applying reusability in testing. Section 3 presents various steps in our reuse-oriented approach to testing, parts of which are automated. Section 4 describes our experience in using this approach in testing a cardiac pacemaker.

2. Reusability in Testing

In testing a family of products, one can reuse test cases, test case requirements, testing techniques and, testing processes across products. In spite of this potential, reusability has not

been widely observed. One of the main reasons is the tight coupling that exists between software and hardware in systems, especially in the case of real-time systems like implantable medical devices that we worked with.

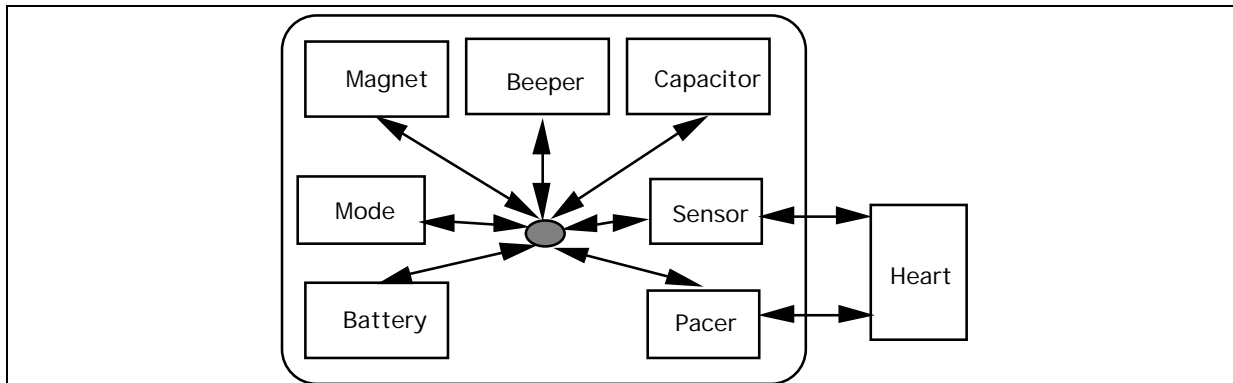
Cardiac rhythm management systems are implantable medical devices used to monitor human hearts and to provide appropriate therapy in case of malfunction. These systems consist of both software and hardware components tightly coupled with one another. Software is constrained by the hardware; the size of the software, the memory and, the programming language used depends on the hardware. A small change in the physical hardware can greatly affect the software architecture. In developing a family of products, the functional requirements may not vary a lot between products, but the hardware requirements could change considerably and they may greatly affect the entire system. For example, a change in the product requirement may dictate a decrease in the size and weight of a pacemaker, which may result in the use of smaller (and different) processing units in it. In such a case, it would be difficult to reuse a test case which is highly dependant on the previous hardware architecture. Most of these test cases must be rewritten.

In such an environment, where a family of products are being developed, the main factor controlling the changes across products is technology. Newer technology dictated enhancements or redesigns in the features of the software. For example, the earlier versions of the pacemaker provided only the pacing feature. Advances in technology allowed sensing of the heartbeat before providing pacing. Significant changes in software architecture were required to add this feature. This change in the software architecture also affected the test cases and code, as they had to be changed to comply with the new design. So even though the requirements for pacing were the same for both the products, the test cases could not be reused.

We observed that due to the coupling and software architecture changes, test case *code* was not a good starting point for reuse. The difficulty in reusing at the code level led us to investigate the right levels for reuse.

On further investigation on the pacemaker products, we found that even though the software architecture changed across products, the major components of the system remained unchanged for every single product. These components were magnet, sensor, pacer, battery, beeper, and, capacitor (Figure 1). This is because the basic requirements for the pacemaker are usually stable as the needs dictated by human physiology remain the same. Further the basic functionality of

these components also remained the same across products, even though the design and implementation was different.



EXAMPLE 1: Sample Pacemaker Description

We describe an existing cardiac rhythm management device that has functionalities to deal with both Tachycardia and Bradycardia pacing [Ellenbogen 92]. The device has several primary components. These components are all inter-dependent on each other for their correct operation. Generic system components are:

- *Magnet*: An external magnet, which when applied to the patient causes the implantable to function in certain ways depending on the current state of the system and patient.
- *Mode*: A feature that controls the device mode changes for the device. The current device mode can be either one of shelf, detect, therapy and sense. Various components can request temporary mode changes concurrently. The Mode feature of this device is responsible for controlling all these requests and ensuring that the mode changes are not contradictory to the permanent device mode (PDM) and the current state of the system.
- *Battery*: A lithium battery is used in the device. The battery has a corresponding software task which is required to take certain actions when it detects that the battery life is below a certain limit or if the state of the battery is such that it may pose a hazard to the patient.
- *Capacitor*: The capacitor is critical to the proper functioning of the device. This is supposed to ensure that in case of shock delivery, the capacitor is charged to the appropriate levels and that the shock delivered is within the acceptable voltage parameters.
- *Sensor*: The critical function of the sensor is to sense the heart beat at both atrial and ventricular chambers. Sensor activates the pacemaker, and sets the appropriate operational mode for the device.
- *Pacer*: This works closely with the mode, and sensor to deliver therapy when sensor activates it.
- *Beeper*: The beeper's main functionality is to give an audible indication of the initiation or completion of operations. In addition, the beeper also sometimes gives an indication of device malfunction.
- *Fault*: The fault task detects any device faults that occur and tasks the appropriate action.

Figure 1. Architecture of a generic pacemaker.

The generic components in figure 1 remain the same across products. We call these components *System Components* (SC). Organizing the test reuse information around these SCs at an abstract level will help in overcoming the tight coupling between the test case and the implementation that hinders reuseability.

3. Our Approach to Testing

Our approach to testing is based on domain level abstractions to decouple the conceptual model from the implementation details, so as to keep test cases architecture independent.

At domain level, we focus on identifying and organizing SCs. SCs are the building blocks of the system, which remain invariant across products. We identified the *operations* of SCs, called System Component Abstractions (see section 3.1) and stored them in a database. These operations are combined into high level description of the test cases called *scenarios* (see section 3.2). These scenarios are then used to generate executable test cases (see section 3.3). Figure 2 shows the overview of this process. Thus, the test case is divided into two parts: a scenario which is written in the domain-level terms and the test code which is generated automatically in the language of the test environment.

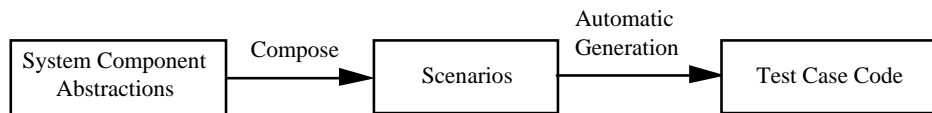


Figure 2: Overview of our approach to testing.

3.1. System Component Abstractions (SCA)

SCAs are domain-level description of the SCs and their related operations. Each SCA has the following attributes.

1. *Name of SC:* Name of the system component that this abstraction represents, such as, magnet, mode, battery, capacitor, sensor, pacer and beeper (see Figure 1).
2. *Operation of SC it represents:* The descriptive names are usually described in domain specific terminology.
3. *Descriptive name for operation:* This is a name for the operation written in domain-specific terms. The advantages of this is that it is understandable by both the domain experts and the test engineers.
4. *Corresponding implementation:* The corresponding implementation of the abstraction will be hardware and programming language dependent.

For example, a SCA for the Mode component of the pacemaker, with three major operations, is:

EXAMPLE 2: SCA for the Mode component.

System Component : **Mode.**

1) Operation 1.

Descriptive name : check mode <device mode>;

Operation : Mode task should support the checking of the current mode of the device.

Implementation :

```

i_ptr = gpib_read( 0x456, 0x3, 0x33 );
i_data = parse_bin( &i_ptr, 0x3, 2 )
if( i_data == 3 )
{
    gpib_write_dec( 34 );
    gpib_write_dec( 35 );
    gpib_write_dec( 36 ); }
data_temp = gpib_read_dec( 5646, 77 );
check_data_ptr = gpib_read(data_temp, 0x67 );
check_data = parse_bin(&check_data_ptr , device mode );

```

2) Operation 2.

Descriptive name : Request PES Induction

Operation : The Mode task can issue a request to the Pacer task for PES induction to commence.

Implementation :

```

gpib_write( 0x0345, 92 );
gpib_write( 0x0090:0091, 89 89 );
gpib_write( 0x9333:0x9338, 00 01 02 03 04 05);

```

3) Operation 3.

Descriptive name : Request Fibrillation Induction

Operation : The Mode task can issue a request to the Pacer task for Fibrillation Induction to commence.

Implementation :

```

i_ptr = gpib_read( 0x3, 0x33 );
if( i_ptr == 0x13 )
{
    gpib_write_dec( 0x34, 1 ); }
gpib_write( 0x0345, 92 );
gpib_write( 0x0090:0091, 89 89 );
gpib_write( 0x9333:0x9336, 00 01 02 03 );

```



SCAs are created using two techniques, namely domain analysis and reverse engineering. Domain analysis is used to identify SCs and their related functions. In domain analysis the domain experts identify the generic system components and their related operations by studying the system and its functionalities. Guidant has many domain experts (medical doctors) who are familiar with the medical devices. It is relatively easy for them to identify SCs and define their major functionalities and to come up with descriptive names for these.

The corresponding implementation of these operations of SCs can be generated by either programming it or reverse engineering it from the existing system code. For reverse engineering the implementation from the existing test cases, we first identified domain variables related to the various SCs and then slice out the relevant source code using program slicing techniques [Joiner 94, Huang 96]. This process enabled us to construct the implementation of the abstractions from the existing software artifacts. Figure 3 illustrates the process that we followed to generate abstractions.

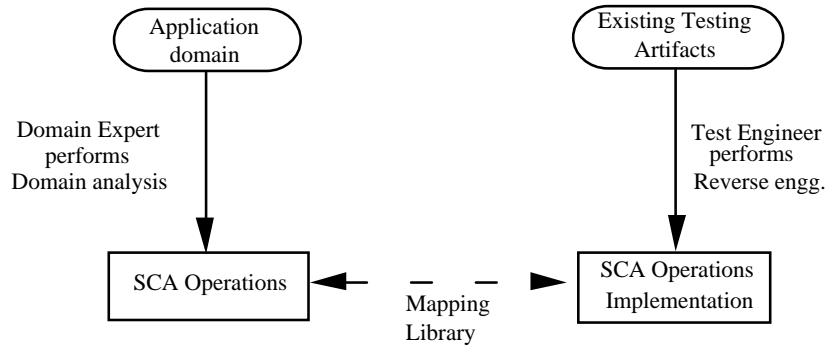


Figure 3: Abstraction generation from existing artifacts and application domain.

The operations in SCA fall into different classes, such as simple, parameterized and boolean. This classification helped in selecting the appropriate abstraction for scenario generation and in classifying different reuse cases, to be described in section 4. *Simple operations* are operations that do not require any parameters. *Parameterized operations* are simple operations requiring parameters. *Boolean operations* are operations which return a Boolean value. Some examples of these are shown in the table 1.

SC	Operations	Type
Magnet	<ul style="list-style-type: none"> Apply magnet for specified time; Remove magnet; 	<ul style="list-style-type: none"> Parameterized Simple
Mode	<ul style="list-style-type: none"> Check current mode; Request PES induction Request Fibrillation induction 	<ul style="list-style-type: none"> Parameterized Simple Simple
Capacitor	<ul style="list-style-type: none"> Charge capacitor to certain voltage; Check capacitor charge Dump capacitor charge 	<ul style="list-style-type: none"> Parameterized Boolean Simple
Beeper	<ul style="list-style-type: none"> Initiate beep at specified frequency for specified time; 	<ul style="list-style-type: none"> Parameterized
Pacer	<ul style="list-style-type: none"> Check induction requestor Id 	<ul style="list-style-type: none"> Boolean
Fault	<ul style="list-style-type: none"> Check System Faults Check Patient Faults 	<ul style="list-style-type: none"> Boolean Boolean

Table 1: Some basic pacemaker system components and their related operations.

3.2. Test Scenarios

Scenarios are high-level description of the test cases. A scenario tests a requirement of the system. Each requirement in the system will have at least one test scenario generated for it. The operations in the various SCAs are combined to form scenarios. The SCA operations are combined using the basic constructs allowed by a scenario description language that we used. The constructs included basic operations, such as sequencing, looping, and conditionals. An example of this shown below.

EXAMPLE 3:

Mode requirement for capacitor charge on mode change:

If the current mode is sense and the magnet is applied for 30 seconds, and if as a result of this, the mode changes to therapy, the capacitor will be charged to 15 volts after the previous capacitor charge has been dumped. Otherwise if the mode does not change to therapy, the capacitor will be charged to 17 volts without a capacitor dump.

The scenarios is generated by the software engineer by choosing the appropriate SCA operations and existing scenarios. For example, to generate the test scenario for this Mode requirement, we selected the SCs such as Mode, Magnet and Capacitor and their operations (see Table 1) as the relevant ones for this scenario. The abstractions that were chosen to test this requirement.were:

Operations chosen
Check current mode <mode>
Apply magnet for <time> seconds
Dump capacitor charge
Charge capacitor to <volts> volts

These abstractions are then combined to form the test case scenario as shown below:

Scenario for Mode requirement	SCA used for operation selection.
Check current mode <mode>; if(<mode> is Sense)	Mode
Apply magnet for 30 seconds;	Magnet
Check current mode <mode>; if(<mode> is Therapy)	Mode
Dump capacitor charge;	Capacitor
Charge capacitor to 15 volts;	Capacitor
else Charge capacitor to 17 volts;	Capacitor

The column on the left shows the scenario that is generated for the mode requirement. The column on the right shows the SCAs from which the operations were chosen.



3.3. Test Code Generation from Scenarios

From the test scenarios, a partial or skeleton code can be generated using the corresponding implementations in the SCA. The partial code can be generated by substituting the operations in the scenario by their corresponding implementations (figure 6). This reduces the effort from going from the high-level scenarios to the actual implementation of the test cases.

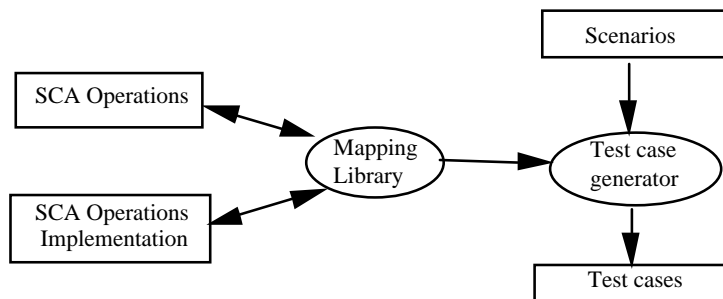


Figure 6: Automatic test case generation from test scenarios using mapping library

Links are maintained between the operations and their implementation in a mapping library. The mapping library is used in the generation of test code for each of the scenarios. Test scenarios are translated into the code automatically using the links between the operations contained in the scenarios and their corresponding implementations.

We now describe the generation of a test case from a test scenario for a Magnet task requirement.

EXAMPLE 4: A requirement and corresponding scenario.

Magnet requirement for setting therapy mode:

If magnet applied on the device for 30 seconds, the mode changes to therapy.

The test scenario for this requirement is.

Scenario for Magnet requirement
Apply magnet <30 seconds> if check current mode <THERAPY> test succeeded; else test failed;

Figure 7. Scenario to test Magnet requirement.



The mapping library maintains two links a) between scenarios and the operations used in them, and b) a link between the operation and its implementation. Figure 8 shows two operations with their corresponding implementations. The link between the operations and their implementations is maintained in the mapping library by associating the unique identifiers for the operations and implementation. In figure 8, the operation denoted by the Op1 identifier has the implementation Imp1, and this association is stored in the mapping library as Op1::Imp1.

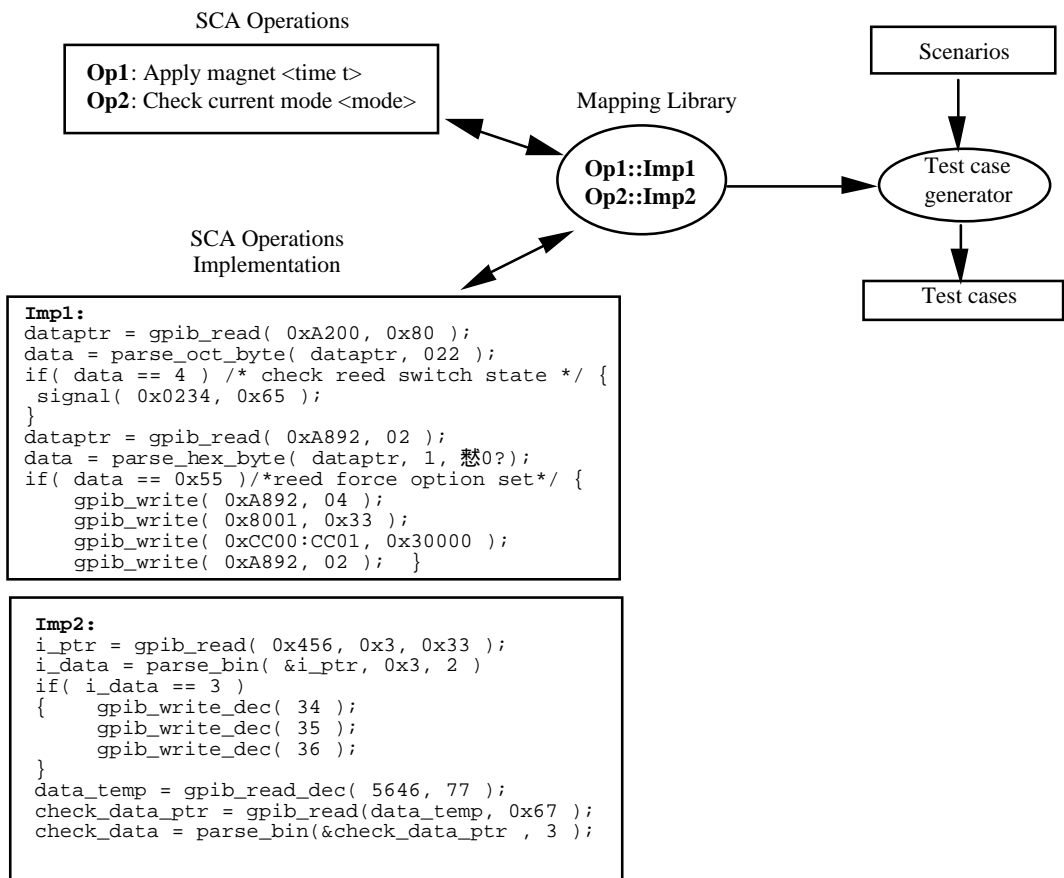


Figure 8: Example of mapping library links between abstractions and corresponding implementation.

To generate the executable test case code for the test scenario, the operations in the scenarios are replaced with their associated code (as stored in the mapping library). The skeleton test case code that is generated for the Magnet task requirement is shown in Table 3. This skeleton test case code can then be edited by the test engineer to obtain the executable test case code. For example, for the skeleton test code shown in table 3, the test engineer will have to complete the condition checks shown in the last two lines, to perform the actions depending on the failure or success of the test.

```

Test case code generated
dataptr = gpib_read( 0xA200, 0x80 );
data = parse_oct_byte( dataptr, 022 );
if( data == 4 ) /* check reed switch state */ {
  signal( 0x0234, 0x65 );
}
dataptr = gpib_read( 0xA892, 02 );
data = parse_hex_byte( dataptr, 1, '\0' );
if( data == 0x55 )/*reed force option set*/ {
  gpib_write( 0xA892, 04 );
  gpib_write( 0x8001, 0x33 );
  gpib_write( 0xCC00:CC01, 0x30000 );
}

```

```

gpib_write( 0xA892, 02 ); }
i_ptr = gpib_read( 0x456, 0x3, 0x33 );
i_data = parse_bin( &i_ptr, 0x3, 2 )
if( i_data == 3 )
{
    gpib_write_dec( 34 );
    gpib_write_dec( 35 );
    gpib_write_dec( 36 );
}
data_temp = gpib_read_dec( 5646, 77 );
check_data_ptr = gpib_read(data_temp, 0x67 );
check_data = parse_bin(&check_data_ptr , 3 );
if( check_data == THERAPY ) { /* test succeeded */
else { /* test failed */ }

```

Table 3: Test case code generated for the Scenario shown in Figure 7.

The advantage of this type of automatic generation is when implementation of any of the operations change, the scenarios are not affected as they are not directly linked to the implementation. And the code for the scenarios, that includes the changed operations can be easily regenerated using the links.

The mapping library can be updated when a new product is being tested. The test designers can identify new test operations. The mapping library can then be updated to reflect the appropriate links.

4. Examples of Test Case Reuse

Modifications in software can occur in different software artifacts during the development of software, all these affect the test case generation and reuse. Changes occur in requirements, design, code, and the underlying implementation platform.

We observed that in case of evolving systems, two kinds of changes were necessary to the operations and scenarios: a) specialization and b) modification. Changes due to *specialization* occur when operations or scenarios are generated by choosing the appropriate subset of existing operations or scenarios (see section 4.3 and 4.6). Changes due to *modification* occur when test operations or scenarios are generated by changing the implementation of existing operations or scenarios (see section 4.2 and 4.5).

These changes are dictated by the changes in the software artifacts, such as requirement, design, code and software environment. We now describe how these different changes affect the reuse of the SCA and scenarios.

4.1. Operation Reuse As Is

Existing operations can be reused as is if the requirement specification or design changes, but if the test cases for the new requirements or design can be specified in terms of the existing test operations. In this kind of reuse, neither the descriptive name nor the representative functionality

nor the implementation require any change. All three types of operations (simple, boolean and parameterized) can be used in this kind of reuse and the type of the operation remains the same after reuse.

EXAMPLE 5:

The pacemaker shown in figure 1 provided only ventricular pacing and therefore the only valid Bradycardia pacing modes were VVI (Ventricular inhibited) and VVR (Ventricular triggered). The next generation of pacemakers incorporated Atrial pacing also. Thus the Bradycardia pacing modes were extended to include in addition to VVI and VVR, VVD, DOO, DVI, DDI and DDD. The existing operations for the VVI and VVR modes were reused as in the new system.

XYZ Bradycardia pacing abstractions	Next generation Bradycardia pacing abstractions
1. Set VVI pulse amplitude.	Reused as is from previous pacemaker
2. Set VVR pulse width	
	Additions
	3. Set DDI pulse amplitude.
	4. Set DOO pulse width

Table 4: Changes in XYZ pacemaker abstractions.



4.2. Operation Reuse with Modification

Operations can be reused with slight modifications to their implementation when changes in the functionality of the SCA dictate changes to the corresponding implementation. In this kind of reuse, the descriptive name of the the operation does not change. However, the functionality that the operation represents and its implementation are changed. All three types of operations (simple, boolean and parameterized) can be used in this kind of reuse and the type of the modified operation may be different from the type of the original operation. For example, the type of the original operation may be *simple* and the type of the modified operation may be *parameterized*.

EXAMPLE 6:

The original and modified operations for the VVI pacing requirement for the Pacer SCA are shown in tables 5 and 6.

Name of SC:	Pacer
Descriptive name:	inhibit ventricular sensing
Function of SC represented:	The ventricular sense circuit inhibits the ventricular pace pulse when the intrinsic ventricular rate is faster than the programmed rate.
Implementation	<pre>float p_rate, i_rate; p_rate = get_prog_rate(); i_rate = get_intr_rate(); if(i_rate > p_rate) { // Original pace_pulse(OFF);</pre>

Table 5: Original VVI abstraction

Name of SC:	Pacer
Descriptive name:	inhibit ventricular sensing
Function of SC represented:	The ventricular sense circuit inhibits the ventricular pace pulse when the intrinsic ventricular rate is faster than or equal to the porgrammed rate
Implementation	<pre>float p_rate, i_rate; p_rate = get_prog_rate(); i_rate = get_intr_rate(); if(i_rate >= p_rate) { // Modified pace_pulse(OFF);</pre>

Table 6: Modified VVI abstraction



4.3. Operation Reuse with specialization

Operations can also be reused from the existing ones through specialization. This kind of reuse occurs when the new operation is a subset of the functionality of the original abstraction. Parameterized abstractions fall in this category. The parameters serve the function of selecting the required subset of the implementation that is to be used. Simple operations cannot participate in this reuse class. In this kind of reuse, the type of the specialized operation is the same as the type of the original operation.

EXAMPLE 7:

One of the operations of the Pacer SC is to provide fibrillation induction. A test requirement for the Pacer SC is as shwon below. Table 7 shows the abstraction specialization that occurs when different scenarios are generated for this test requirement.

Test Requirement 3.z.2

Fibrillation induction should commence when requested and should continue until a) fib abort command is received or b) if sync is dropped for 250 ms or if either c) PES or d) external induction is requested. Any combination of the above fib termination conditions can be specified by the requesting task.

<u>Scenario</u> (Termination conditions are a, b, and c)	<u>Scenario with specialized abstraction</u> (Termination conditions is only c)
if(check induction requestor Id == VALID) start fibrillation induction cancel fibrillation induction(a, b, c) // Original	if(check induction requestor Id == VALID) start fibrillation induction cancel fibrilaltion induction(c) // Specialized

Table 7: Scenarios for test requirement 3.z.2, showing specialized abstractions



4.4. Scenario Reuse As Is

The entire test scenario can be reused when the requirements are reused across systems. We observed that in the case of evolving safety-critical systems, this was the most popular kind of scenario reuse.

EXAMPLE 8:

Figure 1 shows some of the enhancements that were made to the pacemaker shown in table 1. During transitions from version 1 to version 2, only the abstractions could be reused. The scenarios could not be reused in this case as the interaction among the various pacing modes had changed.

For example, in the new system transitions from the VVI pacer mode to the DDD pacer mode and vice versa were possible, whereas in the previous version of the pacemaker such transitions were non-existent. During transitions from version 2 to version 3 most of the scenarios that were generated for version 2 could be reused as the requirements for A and V pacing were reused in version 3.

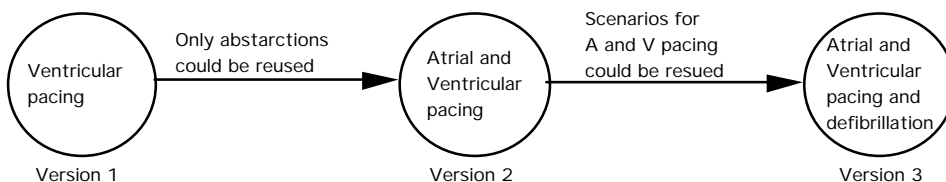


Figure 9: Enhancements to the Pacemaker



4.5. Scenario Reuse with Modification

When only interaction among different components of the system changes, the test scenario may have to be modified while the operations in them are reused as is. A typical case is when the requirement undergoes changes but the new requirement can be satisfied using the existing operations.

EXAMPLE 9:

One of the operations of the Pacer SC is to provide fibrillation, PES and external induction. Different SCs can request induction and the Pacer SC will decide whether any type of induction can be activated. The original and changed requirement for PES induction is shown below. The corresponding scenarios are shown in table 8.

Test requirement 2.x.4

PES induction will commence when requested and only if fib or external induction is not on at that time and will continue until an external abort command is given.

Modified Test requirement 2.x.4

PES induction will commence when requested and will continue until fib or external induction is activated or if an external abort command is given.

Scenario for test requirement 2.x.4	Scenario for changed test requirement 2.x.4
<pre> if(check induction requestor Id <VALID> AND check induction start <VALID> { start PES induction ; } while (1) { </pre>	<pre> if(check induction requestor Id<VALID>) { start PES induction; } while(1) { if external_abort AND </pre>

if external abort { stop PES induction; } }	check induction start<NOT VALID> //Scenario modification { stop PES induction; } }
--	---

Table 8: Scenarios for original and new test requirement 2.x.4



Usually, both the scenarios and operations may need modification. This can occur when the functionality of the components is changed (*operation change*) and the interaction among the various components changes (*scenario change*). The changes to the operations can be either made through specialization or modification reuse.

EXAMPLE 10:

One of the operations of the Pacer SC is to provide fibrillation, PES and external induction. The original and changed requirement for fibrillation is shown below. The corresponding scenarios are shown in table 9.

Test requirement 3.x.4

Fibrillation induction should commence when requested and if a) PES induction is not on or b) if external induction is not on or c) if therapy was not aborted previously. Any combination of the above commencement conditions can be specified by the requesting task. Fib induction will continue until an external abort command is received. This test requirement was modified so that fib induction should commence only if condition c) is satisfied, and fib induction will continue until an external abort command is received or until either of a, b and c are activated.

Scenario for test requirement 3.x.4 if check induction requestor Id<VALID> AND fibrillation start requirement (a, b, c) { start fibrillation induction; } while(1) { if external abort { stop fibrillation induction } }	Scenario for modified test requirement 3.x.4 if check induction requestor Id<VALID> AND fibrillation start requirement (c) // Abstraction specialization { start fibrillation induction; } while(1) { if(external abort AND fibrillation stop requirement (a, b, c)). // Scenario <i>modification</i> { stop fibrillation induction; } }
---	--

Table 9: Scenarios for original and new test requirement 3.x.4



4.6. Scenario Reuse with Specialization

This occurs when scenarios are reused by choosing the required subset of the existing scenario. This kind of reuse occurs when the new requirement or design is a subset of the original requirement or design.

EXAMPLE 11:

One of the operations of the Pacer SC is to provide fibrillation, PES and external induction. The original and changed requirement for fibrillation is shown below. The corresponding scenarios are shown in table 10.

Test requirement 23.x.2

External induction should commence when requested and if a) PES induction is not on or b) if fib induction is not on or c) if therapy was not aborted previously. External induction will continue until an external abort command is received. This requirement is modified so that external induction will commence only when conditions a) and b) are specified. Although this can be captured by abstraction specialization, we illustrate an example where the scenario may have to be specialized.

Original Test Scenario	Specialized Test Scenario
<pre>if check induction requestor Id<VALID> AND PES induction <NOT ON> AND fibrillation induction <NOT ON> AND therapy abort <NOT VALID> start external induction; } while(1) { if(external abort) { stop external induction; } }</pre>	<pre>if check induction requestor Id<VALID> AND PES induction <NOT ON> AND fibrillation induction <NOT ON> AND // Scenario specialization start external induction; } while(1) { if(external abort) { stop external induction; } }</pre>

Table 10: Scenarios for original and new test requirement 23.x.2

5. Illustration

Now we describe a situation where functional test case reuse was performed, in order to illustrate how the proposed reuse-oriented approach handles the major changes in implementation.

Functional Test Case Reuse: During simulation/testing of the cardiac pacemaker all interactions between the raw register programmer with the heart simulator were done through the RS232C serial interface. The test signals and probes passed over the bus are specified as: *rs232(value, memory location, read/write)*.

Later, this interface implementation of the test station was upgraded to use parallel IEEE-488 General Purpose Interface Bus. This implied a wide variety of changes to the signal and probes usage in the various test cases. But as our test scenarios were not directly dependent on the implementation of the system of the test environment, we could generate unit test cases using the all the existing test scenarios because none of the functionality of the product changed. An example of this is shown in example 10.

EXAMPLE 12:

Test Requirement for Query Patient Warning Function: Test the fault service function which determines whether or not an enabled patient warning has occurred. Enabled patient warning faults are set in the fault parameter block, the addresses of the faults are specified in device telemetry interface specification.

Figure 10 shows the patient warning faults, their corresponding interface addresses, the test scenarios for the test requirement of Query Patient Warning fault service function in Guidant's devices, test code before and after the change in the interface from RS232C to IEEE-488 GPIB. To handle the change in the interface, only the change in the implementation of individual SCA's operation implementation were changed, but the test scenarios were used as is. After changes in the implementation were made, the test cases were automatically generated from the scenarios.

Fault	Address	Test Scenario	Original test code	Changed test code
-------	---------	---------------	--------------------	-------------------

		(Domain-dependent)	(RS232C dependent)	(GPIB dependent)
Cap charge	0x0004	reset cardiovert and	rs232(0,0x0004,write)	gpib(0,0x0004, write);
Atrial supply	0x0008	system faults	cap_dump();	cap_dump();
Ventricular supply	0x0010	set cap charge fault	rs232(2,0x0004,read	gpib(2 ,0x0004, read);
Voltage supply	0x0040	check cap form fault)	gpib(cc, 786, 2)
Charge time-out.	0x0080	status	rs232(cc, 786, 2);	if(cc) return true;
Oscillator deviation	0x2000	return result of test	if (cc) return true;	else return false;
Checksum fault	0x4000		else return false;	
Battery Post ERI	0x8000			

Figure 10: Example of Functional test case reuse



Thus, this example showed that one can achieve high reuseability across products even when significant changes are made to the test environment or implementation platforms. This is possible because the high-level test scenarios capture the implementation-independent knowledge of the test cases. Only the mapping library need to be changed, but the scenarios are reusable as is. Without any high-level abstractions like scenarios, it would have required considerable work on part of the tester to generate all the test cases confirming to the new system implementation.

6. Experience

Speedup: Using the reuseability approach discussed we achieved a significant speedup in the testing process. The use of high-level abstractions (SCAs) reduced the effort required to go from the initial domain-level design of the test case to the test case implementation. The use of the mapping library reduces the effort required to go from the scenarios to an executable implementation of the test case. The percentage of reuse was quite high as depicted in figure 11.

In general there will be a variety of changes in the system requirements, design and code, and so a combination of various reuse classes (section 4) must be used in practice. In our case study at Guidant, we found there was a high percentage of reuse in the “As Is” category both for operations and scenarios. The figure shows the percentage reuse in the different categories.

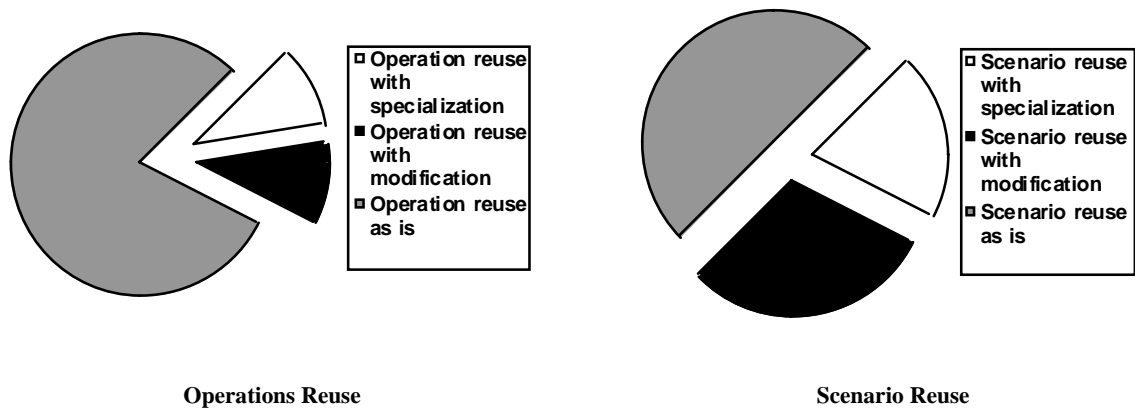


Figure 11: *Distribution of reuse in various classes.* In moving from one product to another within the same family of products, we found that for the Magnet, Mode and Fault components of the pacemaker (figure 1), 90% of the operations could be reused. From these operations that could be reused, 80% of them were reused as is, 10% were reused with modification, and 10% with specialization (see Figure 6). Also, 70% of the scenarios were potentially reusable, out of which 50% were reused as is, 30% of those scenarios were reused with modification, and 20% were reused with specialization.

Inter Testing Stage reuse: Testing is a multi-stage process. A typical system consists of software, hardware and firmware components. First unit testing is performed for each component followed by an integration test. We found that most of the test scenarios from the unit and component testing stage can be reused in the integration testing stage, with only the mapping library between the abstraction and implementation being changed.

7. Conclusions

This paper proposed an abstraction based approach to test case reuse. Interestingly, the scenarios and operations specified can also help in requirements and design reuse. However the applications of these in an industrial environment is a challenging task due to the complexity of industrial software [Tsai 95, Onoma 95]. We found this approach by recognizing the invariant parts of the software systems. The key idea is to abstract out the domain-level operations and support test case development that depends on these operations. The SCA are at the domain level and can thus be understood and used easily than the actual test code. The approach decouples the conceptual model from the implementation details. Also, modularity is achieved by having the abstractions centered around system components, and by keeping them separate from the implementation providing information hiding.

Operational test scenarios can be derived by composing and combining various operations at domain, application and programming levels. This assists in the rapid development of test cases as the test developer can design the test cases at domain or requirements level, thus reducing the effort required.

We found that, even though this reuse approach to testing is generally useful, it is especially useful for those companies with one or more of the following characteristics:

1. Companies that develop a family of similar products, and when the products are evolving from one another, because in this case many objects of the system remain same across products and can be reused easily;
2. Companies that develop mission-critical, safety-critical, or real-time systems as they need to invest a large amount in the testing and revalidation of their software, and reuse helps in the reduction of the cost associated with testing such systems;
3. Companies that develop products that are under constant evolution as the market or technology changes, as again the higher level abstractions help cushion the changes in software technology and architecture.

Changing system requirements can be economically accommodated by modifying incrementally scenarios, operations and implementations respectively. Further these changes can be made independent of one another as each of them have been developed modularly.

Conceptually this approach to reuse is simple, but to implement this in a real project which had hundreds of thousands of lines of code, recognizing the commonalities among the test cases, and implementing a mechanism for systematic reuse is a huge task. We applied our reuse techniques at the testing stage of a real project that involved the development of a cardiac rhythm management system, and have presented results from this project. We obtained significant improvements in the effort required to test systems.

8. References

- [Ellenbogen 92] K. A. Ellenbogen, Ed., *Cardiac Pacing*, Blackwell Scientific Publications, Massachusetts, 1992.
- [Huang 96] H. Huang, W. T. Tsai and S. Subramanian. "Generalized Program Slicing", Technical Report, University of Minnesota, 1996.
- [Joiner 94] J. Joiner. "Data Centered Program Understanding", PhD Thesis, Dept. Of Computer Science, University of Minnesota, 1994.
- [Onoma 95] A. K. Onoma, W. T. Tsai, F. Tsunoda, H. Suganuma, and S. Subramanian, "Software Maintenance -- Industrial Experience", *Journal of Software Maintenance*, Dec. 1995.

- [Elliott 94] L. Elliott, R. Mojdehbakhsh and W. T. Tsai, "A Process for Developing Safe Software", Proceedings of the Seventh Annual IEEE Symposium on Computer-Based Medical Systems, IEEE CS Press, Los Alamitos, California, Jun. 1994.
- [FDA 89] *Preproduction Quality Assurance Planning: Recommendations for Medical Device Manufacturers*, Office of Compliance and Surveillance, Center for Device and Radiological Health, FDA, September 89.
- [FDA 91] *Reviewer Guidance for Computer Controlled Medical Devices Undergoing 510(K) Review*, Office of Device Evaluation, Center for Device and Radiological Health, FDA, August 91.
- [Mojdehbakhsh 94a] R. Mojdehbakhsh, W. T. Tsai, S. Kirani, and L. Elliott, "Retrofitting Software Safety in an Implantable Medical Device", IEEE Software, No. 1, January 1994, pp. 41-50.
- [Mojdehbakhsh 94b] R. Mojdehbakhsh, S. Subramanian, R. Vishnuvajjala, W. T. Tsai, and L. Elliott, "A Process for Software Requirements Safety Analysis", Proc. Intl. Symposium on Reliability Engineering, Nov. 1994.
- [Tsai 95] W. T. Tsai, L. Elliott, S. Kirani, R. Mojdehbakhsh, T. Sano, S. Subramanian, S. Uehara, "Joint University-Industry Research Projects in Software Engineering - Perspectives and Experience", TR 95-057, Department of Computer Science, University of Minnesota, MN 55455.