

Design for Constraint Violation Detection in Safety-Critical Systems

Satish Subramanian^{*}, Wei-Tek Tsai^{†*} and Sanjai Rayadurgam[†]

^{*} *Cardiac Pacemakers Inc., Guidant Corporation
4100 Hamline Avenue North, St. Paul, MN 55112.*

[†] *Computer Science and Eng., University of Minnesota
200 Union Street SE, Minneapolis, MN 55455.*

Email: tsai@cs.umn.edu

Abstract

In safety-critical systems, certain safety constraints must be satisfied before an operation can be performed. Such constraints typically depend on the state of the system at the instant of invocation of the operation. Further, for a specific version in a family of related systems, its intended application and the individual user profiles may play a role in determining the constraints. To promote reuse while allowing customizability, a good design should decouple the operations from their associated constraints. Also, the increased flexibility should not entail significant execution time and memory overheads. This paper presents one such design of an object-oriented framework for constraint verification and applies it to our motivational problem in the context of implantable cardiac devices. It also serves a broader objective of cataloging the experience gained in developing safety-critical systems. Such a knowledge base will be of practical value to the developer community.

1. Introduction

Pacemakers and defibrillators, called Pulse Generators (PG), are implantable medical devices that continuously monitor the human heart and deliver appropriate therapies to correct cardiac arrhythmia [1]. A PG contains a capacitor, a battery, a CPU, computer memory and firmware. Hundreds of parameters determine the operating characteristics of a PG. The physician, based on the medical needs of the patient, determines the values for these parameters. These values are programmed into PG using a device called Programmer. It consists of a user-friendly graphical user interface, software to verify that the parameters entered by physicians are safe and consistent, and telemetry devices for communicating with a PG. Both PG and Programmer are safety-critical systems, and must meet the guidelines of various regulators (such as FDA and ISO) for reliability, safety and documentation standards. Cardiac Pacemakers Inc.

(CPI) of Guidant Corporation develops such medical devices. These devices share many features and functions within and across product families. New versions of these products often enhance or modify certain functions while inheriting the rest of the characteristics from earlier devices. Object-oriented techniques are particularly suitable for such development.

Safety can be retrofitted [4] after building a system based on its functional requirements. But it is necessary to employ design techniques that are tailored to meet the characteristics and constraints of safety-critical systems. Object-oriented techniques such as incremental modeling, reuse and abstraction have been recommended in the literature to address these issues. Some design notations and methodologies have been proposed for developing critical systems [7]. A repertoire of prescriptive design solutions for specific design problems, such as design patterns [2], [8] would complement this. The experience gained in developing safety-critical systems thus can be cataloged and reused. Our current work is a step in that direction. This paper presents a solution for constraint violation detection in safety-critical systems. The features of this design are elucidated with a specific example in the implantable medical device domain.

Constraint verification is a mechanism to ensure that the conditions required to execute an operation are met. In the context of safety-critical systems, non-satisfaction of constraints may result in serious hazards, which may lead to loss of property and lives. Many safety-critical systems are reactive systems. These systems monitor the inputs received from sensors that measure various parameters of the process being controlled and in turn send outputs to actuators that control the process. Examples of such systems include control systems for nuclear reactors, fly-by-wire (auto-pilot) avionics systems, automobile control features like cruise-control, and cardiac medical devices like pacemakers and defibrillators. These systems have a *controller* that monitors a process through *sensors* and manages it through *actuators*. For cardiac devices, the human heart

function is the controlled process, various leads are used as sensors and actuators, and the implanted device is the controller. Inputs from the sensors continually update the controller's view of the current state of the process. The controller sends commands to the actuators as and when it is deemed necessary to change the state of the process. The controller, the sensors and the actuators are all safety-critical. Faulty sensors or actuators can damage the communication between the controlled process and the controller. Also, the controller should not initiate actions at an inappropriate moment. For example, sending a shock to the heart when it is beating at a high rate due to physical exercise is likely to traumatize the patient rather than correct any cardiac mal-function. Such an operation by the controller is considered hazardous. Typically, the controller monitors the process and collects different inputs over a period of time before deciding to act on an apparent mal-function. Before initiating such an operation various constraints may have to be satisfied. For example, it may not be appropriate to shock the heart in rapid succession. The PG must check if a certain number of shock therapies have already been sent in the last x seconds and if so defer further therapy.

Another dimension to this problem is in configuring the controller. The design of the controller takes into account various factors, like the intended functions of the controller, the characteristics of the process to be controlled, and the type of sensors and actuators used. Often the controller is designed with a number of configurable features called parameters. Depending on the environment in which a particular controller is to be used, domain experts set the parameters to specific values. These parameters then determine the behavior of the controller. Except for some simple systems, these parameters are inter-dependent. It is not meaningful and often hazardous to set each parameter independently to any desired value. So, the values assigned must be verified for safety and consistency before committing the controller to a specific configuration. Thus, the process of setting the parameter values in the controller is in itself a safety-critical function [11]. For cardiac devices, a Programmer is used to set the configurable parameters.

The design of the constraint verification component of the Programmer is the theme of this paper. While it is possible to design a new Programmer for each device completely from scratch, it is not economical. From the manufacturer's viewpoint, it is important to reuse existing components. Development efforts are typically incremental. Reuse, modular design and localization of impact due to changes are of high priority [9]. Thus we address this design issue with two considerations in mind. One is the safety-critical nature of the Programmer and the other is the need of reuse.

The rest of the paper is organized as follows. Section 2 presents in detail a problem that arises in the design of

Programmers. Section 3 formulates the problem in general terms and highlights the design issues. Section 4 discusses the factors that influence the design and then presents an object-oriented design that takes these factors into consideration. It also describes how this solution is used to address our motivational problem. Section 5 analyzes the implications of adopting the design to verify constraint satisfaction. Section 6 concludes the paper.

2. Setting parameters in a Pulse Generator

Physicians enter parameter values with the aid of a Programmer. Incorrect values may be entered by mistake. So the programmer must verify a host of constraints on the values before writing them on to the PG. The type of the device and the interaction between the different parameters determine the constraints. Further multiple devices across different product families share the same set of parameters, but the constraints depend on the specific device type. The following example illustrates how various parameters interact in a safety-critical device. The example highlights both the commonality and the differences among different instances of the problem.

Pacing Range: A Parameter interaction example. The pacing rate of a PG is typically restricted to a range because human heart can not beat too fast or too slow. The Lower Rate Limit (LRL) specifies the lower bound of the range, and the Upper Rate Limit (URL) the upper bound. These two parameters do interact. The actual range for a specific device is determined by a number of factors including the patient's medical needs. Both these limits are programmable by physicians. A simple but important rule concerning these two limits is that $LRL < URL$. If this rule is violated the pacemaker will not pace the heart when it is required to do so. This is a serious hazard. The Programmer must verify that the pacing range is valid before programming the values for LRL and URL into a PG. This rule may vary depending on the context.

First, the interaction rule changes with different products. For a specific device, say PG_1 , its interval resolution requirement may dictate that the range is at least 20 beats. This means that, for the device to effectively differentiate between two heart beat rates, they should be separated by 20 beats. Thus the valid pacing range constraint for this device is, $LRL \leq URL - 20$.

Second, parameter interaction rules may also change depending on the context in the same device. For example, in tracking modes $VRP < MTR - 90$, and in non-tracking modes $VRP < LRL - 90$, where VRP represents the ventricular refractory period. Thus, different sets of constraints apply for a single parameter even in the same device.

Finally, not all the parameters will be implemented in

the device in a uniform way. Some may be implemented as special purpose registers and some as a combination of multiple registers set to certain values. Thus accessing or setting a value for a parameter in the PG may be done differently for each type of parameter. Any design solution should accommodate all these possibilities. The next section abstracts the key ingredients of this example and highlights the design issues.

3. Constraint satisfaction problem

Suppose that a system consists of many interacting components and certain operations can be performed only after some conditions are satisfied by the system. Further, the conditions that must be satisfied may vary depending on the specific system or even dynamically within the same system based on its state, while the components and the operations remain the same. The design issue is "how do we design the constraint verifiers for the operations such that the operations are independent of the specific constraints which may vary from one context to another?".

3.1. Conditional start-up constraints

These constraints are conditions that must be satisfied before an operation starts execution and are based on the states of the objects in the system. The parameter interaction problem discussed earlier is one such example. Each operation sets some parameters to specific values in the PG, and the constraints depend on the parameters. Assuming the Programmer can set each parameter to any value independent of other parameters, where will the constraint be implemented? A simple design, as shown in Figure 1, is to treat each parameter as a separate object with a *set_value* method to set the value in the PG. Each *set_value* method also has a guard condition. The *condition()* is another method of the object which checks the values of the related parameters before writing these on to the PG. But the implementation of *condition()* must be different for each parameter. So, each instance of parameter will have its own class to implement the startup constraint on the *set_value()* operation. The condition checking assumes that instances of the relevant parameters can be accessed using the *get_param()* method. In general, the constraints in each of the above classes can be implemented using state-based notification schemes as discussed in [3]. But the parameter classes themselves encode the constraints thus making them less extensible and more susceptible to subclass proliferation. Such design issues are summarized in the following sections.

```
//abstract base class for all parameters
class Parameter {
public:
    // constructor - initialize and
    // register parameter instance
    Parameter(ParamID myid) :
        Id(myid), val(get_device_param(my_id))
    {
        Register_param_instance(id, this);
    }

    // to set parameter value
    bool set_val(ParamVal new_val) {
        if (condition() &&
            set_device_param(id, new_val)){
            val = new_val;
            return true;
        }
        else return false;
    }

    // to access parameter value
    ParamVal get_val() { return val; }

    // override this in the derived class
    virtual bool
        condition(ParamVal new_val) = 0;

private:
    // parameter data
    ParamID    id;
    ParamVal   val;
};

// A derived class for parameter LRL
class Param_LRL : public Parameter {
public:
    // constructor
    Param_LRL() : Parameter(LRL_ID) {}

    // verify the rule: URL - LRL > 20
    bool condition(ParamVal val) {
        return (get_param(URL_ID)->
                get_val() - val) > 20;
    }
};
```

Figure 1: Design A – Separate class for each parameter

3.2 Design issues

Avoiding inheritance anomalies. In object-oriented design, proliferation of sub-classes can make the class hierarchy huge and unmanageable. This is called *inheritance anomaly*. Suppose one had a generic base class with a method for the operation and each derived class incorporates a different start-up constraint as in Design A. The class hierarchy grows with each new constraint as Figure 2 demonstrates.

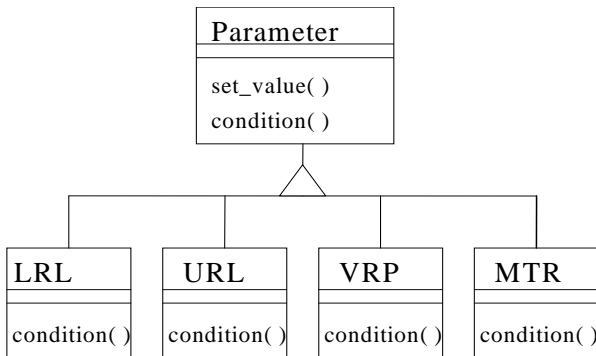


Figure 2. Subclass proliferation

If each type of device to be configured by the Programmer had a different set of rules then a separate set of derived classes are required for each device. This will be a maintenance nightmare. A good design should avoid such proliferation of sub-classes. It should anticipate such scenarios and provide hooks to incorporate different constraints without extending the class hierarchy.

Reuse across product families. Often medical devices are produced as families of products where product requirements and design overlap extensively. Consequently, design should consider reuse and incorporate features to maximize reuse. The interaction rules for parameters for certain operations may depend on the specific product but the operations may be common to many product families and can be reused.

Ease of extensibility. For reuse to be practical, the initial design should identify commonality and make it possible to use this common base for other products. It should clearly delineate the constant part of the design, e.g. parameter, that is common to many products from the variable part of the design, e.g. constraint, which changes with each product.

Change Localization. When reusing, changes may be required to adapt the design to the new context. If the impact of changes were extensive, the cost of performing impact analysis and regression testing would outweigh the benefits, making reuse unattractive. To avoid such a situation, the design should minimize the impact of changes. Consequently, a modular design with different objects interacting through well-defined interfaces is needed.

4. Constraint satisfaction design

This section illustrates an object-oriented design for the constraint satisfaction problem which addresses the issues

discussed earlier. Figure 3 is an object diagram for this design.

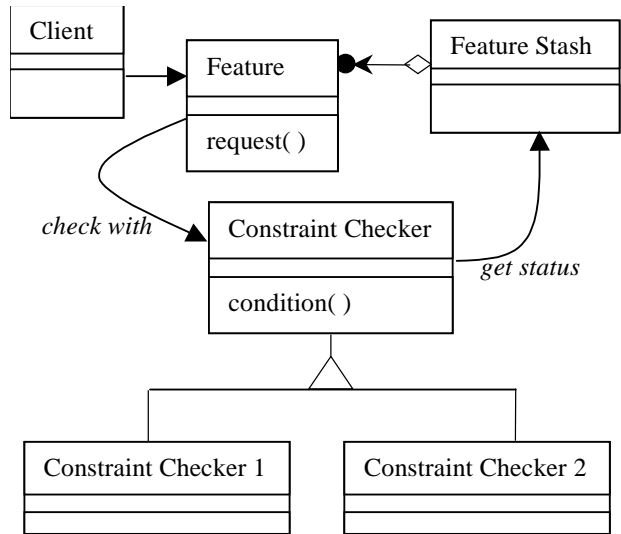


Figure 3. Constraint checker

A *Client* object requests a *Feature* to perform a specific operation by calling its *request()* method. The *Feature* object in turn checks with a *ConstraintChecker* object by invoking its *condition()* method to see if the constraints for starting up the operation are satisfied. But to verify the condition, the *ConstraintChecker* may have to inspect the state of *Feature* objects in the system on which the condition depends. For this the *ConstraintChecker* consults the *FeatureStash* which has access to the states of all instances of *Feature* in the system. Each instance of *Feature* is assigned a unique name or ID and the *ConstraintChecker* uses this unique identifier to query its state. State changes to *Feature* objects are notified to the *FeatureStash*.

In the parameters interaction example, the *Feature* objects are parameters, the *request* operation sets the value of the parameter, the *FeatureStash* has access to the implementation of all parameters, the *condition* method of *ConstraintChecker* enforces interaction rules and the *Client* is the user interface module of the Programmer that triggers these operations in response to user input.

4.1. Design Highlights

Access to the features. The *ConstraintChecker* accesses the states of *Feature* instances through a *Stash*. The *Stash* is a mapping from the feature IDs to associated data. As noted earlier, all parameters may not be implemented in a uniform way. The *Stash* is a convenient mechanism to hide such differences in the structure and implementation of the parameters from other parts of the system. Figure 4

shows the object diagram for a stash design.

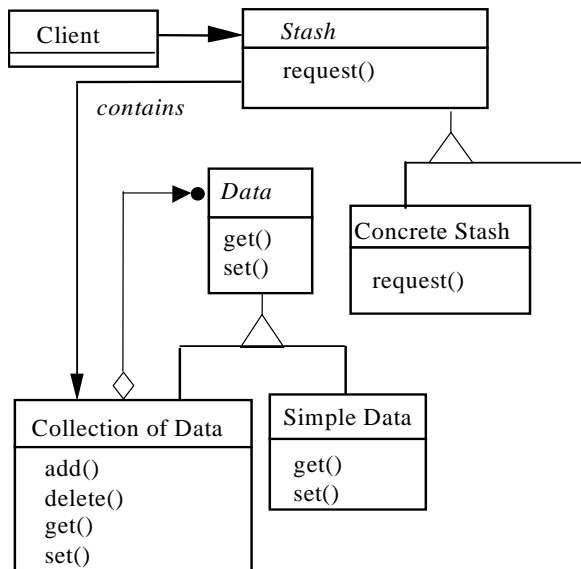


Figure 4. Stash design

The intent of *Stash* design is to act as an interface between the consumers of data and its internal storage and representation. It shields the consumers from the complexity involved in accessing the data. An important implication of this design is that the data and the consumers can evolve independent of each other. In the Programmer example, data objects may actually set and read values in the PG. Thus, depending on the device types and the parameters involved, one may reuse the same implementation for *Data* when the parameter implementations are similar and define new structures when they are different.

Different implementations of constraint checkers. The conditions to be verified could be system-specific and hence may not be reusable across different systems. Different concrete *ConstraintChecker* objects handle different sets of constraints. For the parameter interaction example, a new *ConstraintChecker* object may be used for each type of device that is to be configured by the Programmer. Another aspect of the design is the flexibility in the implementation of these constraint checkers. Simple checkers may be directly hard-coded with the conditions to be verified. But for complex systems like pacemakers, the set of constraints may be stored in a separate configuration file or a database. Such an organization is helpful in many ways. It captures all constraints related to a device in one place. It allows the configuration file to be version managed independent of the code. During development and testing, faults discovered may have to be mitigated by modifying some

constraints. In such cases, the constraint configuration file can be modified without recompiling the source code.

Types of conditions. The condition implemented in the *ConstraintChecker* can be of different types. Each class of conditions must be handled differently.

1. *State of other Features:* If the condition depends on the current states of other features, then the *ConstraintChecker* must query the stash. In a multi-threaded implementation, different threads may request changes to Feature states. If one thread queries a feature state while another is modifying the *ConstraintChecker* may return an incorrect result. A solution is to lock the *FeatureStash* so that only one thread can gain access to modify or query feature states at any given time. While this can unnecessarily block a totally unrelated thread from accessing the *FeatureStash* it avoids the deadlock problem that may occur if locks are used on individual *data* objects in the stash.
2. *History of state of objects:* Sometimes the conditions may depend on the history of changes that happened to some features. In these cases the *Stash* should be able to store the history of changes and also get different versions of a feature as requested. Multi-version data structures can be used to implement such a stash.

Object collaboration. The diagram in Figure 5 shows the message exchange in relative time ordering between collaborating objects in this design. *Feature* objects register themselves with the stash. When a client requests an operation on a specific *Feature* it delegates verification to the *ConstraintChecker* which in turn queries the stash. The *ConstraintChecker* then computes and verifies the constraint and returns the status to the *Feature*. If the condition was satisfied the *Feature* requests the *Stash* to update its state. Finally the stash updates the actual data.

Binding constraint object and its subjects. During system operation the Features need to know their *ConstraintChecker* objects, and vice versa. There are several ways to impart this knowledge to either of them. One way to create this binding is to let the instances of *Feature* and *ConstraintChecker* register themselves with each other during creation. The order of instantiation of the *Feature* and *ConstraintChecker* objects affects this registration. This order depends on the constraint being modeled and the information required by the *ConstraintChecker* from the *Features*. Different cases are:

1. *During Feature creation:* The binding between *ConstraintChecker* and *Feature* objects can occur during the instantiations of *Feature*. The *ConstraintChecker* is passed to the *Feature* instances during creation. Constructor for *Feature* is

overloaded to include the *ConstraintChecker* as a parameter. The default constructor is still available, which makes it possible to configure features without constraint checking too. The *Feature* then (a) registers itself with the *ConstraintChecker* passed to it and (b) saves a reference to the *ConstraintChecker*. This option can be used when the constraint being modeled is generic and the *ConstraintChecker* can have generic references to the objects.

2. *During ConstraintChecker creation:* In this case the features are passed to the *ConstraintChecker* object during its creation. It then registers itself with the *Feature* instances.

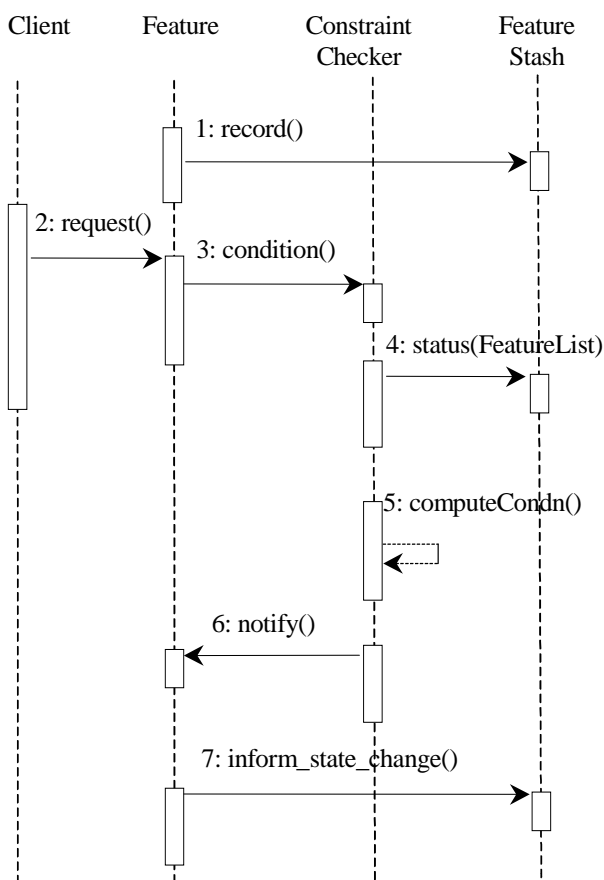


Figure 5. Object interactions

4.2. Application to parameter interaction problem

This section shows how the above design can be implemented to address the parameter interaction problem discussed in Section 2. For brevity we consider only two parameters, LRL and URL. The condition is that, LRL and URL should not be assigned values whose difference is less than 10 units. The C++ code in Figure 6 illustrates this case. In this example, the binding between constraint

checker with the parameters takes place at parameter instantiation time. The constraint checker in turn ensures that the parameter is registered with the appropriate stash. For simplicity, parameter stash is shown as a global variable. A single parameter class is defined and all parameters are instances of this class. The concrete constraint checker *DeviceA_CC* queries a database by constraint id to extract all relevant constraints and then evaluates it. Parameters simply call the condition method of their constraint checkers by passing their ID and the value to be used for the parameter to verify the constraint.

5. Implications of using the design

Let us consider how the use of the above solution addresses the issues that we set out to tackle in the first place. These issues have a direct bearing on the reliability and safety of high-assurance systems [10]. To gain a better understanding, we compare the design B described in the preceding section, with design A that uses a separate class for each parameter.

Avoids inheritance anomalies. Design A as noted earlier suffers from inheritance anomalies. This does not happen in design B. The constraints, the parameters and the implementation of the parameters are all separate. Each parameter is an instance of parameter class. Parameters that have a similar implementation are instances of the same Data class inside the stash. Thus, new structures are defined only when required, and not for each constraint.

Avoids ripple due to changes. Consider the following:

- (1) *Changing a rule:* Suppose the constraint $URL - LRL > 20$ is changed to $URL - LRL > 30$. This affects the two classes *Param_LRL* and *Param_URL*. The *condition()* method in both the classes have to be updated in Design A. In design B, the changes may or may not cause any ripple at all depending on how the conditions are implemented. If the constraints are hard-coded, then it will require a single change in the condition method. If the constraints are stored in a configuration file or in a database, the change affects only that file and not the implementation. We may consider the constraints to be verified as input to the Programmer. The change thus affects only the inputs.
- (2) *Addition of a new rule:* Suppose a new rule, e.g. $VRP < LRL - 90$ is to be added. In design A, this would require a change to the *condition()* methods. In design B, the change again does not affect the implementation if constraints are stored in configuration file or a single change in the *condition()* method if constraints are hard-coded.

Table 1 summarizes the impact of these changes.

```

extern Stash param_stash; // global stash for simplicity

class ConstraintChecker { // The generic constraint checker base class
public:
    virtual bool condition(ParamID id, ParamVal val) = 0; // implemented by derived class
    void register(Parameter *p ) { param_stash.register(p); }
};

class Parameter { // Parameters are instances of this class
public:
    Parameter(ParamID id, ConstraintChecker *cc) : id(myid), cc(mycc), val(-1) {
        cc->register(myid, this); // register with constraint checker
    }

    bool set_val(ParamVal val) { return cc->condition(id) && param_stash.set_val(id,val); }

    ParamVal get_val() { return param_stash.get_val(id); }

private:
    ParamID id; // my parameter id
    ConstraintChecker *cc; // my constraint verifier
};

class DeviceA_CC : public ConstraintChecker{ // A concrete constraint checker
public:
    bool condition(ParamID id, ParamVal new_val) {
        ConstraintSet *cons = get_from_database(id); // query constraint DB on param ID
        bool result = true;

        // check conjunction of all relevant constraints for this parameter
        for(Constraint *c = cons->first(); c != cons->last(); ++c) {
            for (ParamID *p = c->first(); p != c->last(); ++p) {
                if (*p == id) c->use_val(new_val);
                else c->use_val(stash.get_val(*p));
            }
            result = result && c->eval();
        }
        return result;
    }
};

Constraint database has an entry: "URL_ID MINUS LRL_ID LESS_THAN 20"
which is parsed by the get_from_database function

```

Figure 6: Design B – Separate Constraint Checker with a Parameter Stash

Table 1. Impact of changes

<i>Modification</i>	<i>Design A</i>	<i>Design B</i>
Change rule: LRL – URL > 20 to LRL – URL > 30	Change <i>condition()</i> methods in <i>LRL_param</i> and <i>URL_param</i>	Change <i>ConstraintChecker</i> (none if database or configuration file is used)
Add rule: VRP < LRL – 90	Change/add <i>condition()</i> of <i>VRP_param</i> and <i>condition()</i> of <i>LRL_param</i>	Change <i>ConstraintChecker</i> (none if database or configuration file is used)

Configurable. An important aspect of this design is the configurability of the constraints. When constraints are stored in a database the same implementation of a *ConstraintChecker* can be used in multiple contexts by providing the appropriate constraint database as input. Such an option is not available with design A. Further, if we want different constraints to be verified depending on the device being programmed, with design B, it is a simple matter of using the right constraint database or using a different concrete instantiation of a constraint checker. In design A, this would mean defining multiple parameter classes even for the same parameter, just because their constraints are different.

Regression testing. When changes are made to the implementation, the software must be regression tested to

ensure that it did not adversely impact the functionality [5], [6]. Typically, some traceability information between the test cases and the software components or paths being test is maintained. This information is used to select test cases that must be rerun to validate the changed software. In design B with configuration file for constraints, changes to constraints imply changes to the input configuration file. But in design A, changes require modifications to the structure of the code. This has an important implication for testing. When the structure of the code is modified, the number of test cases that get selected for regression testing are typically more than the number of test cases selected when the input alone is modified. Thus design B saves significant testing resources and time.

Reusability. Design B is structured so that reuse is easy and natural. Addition of new constraints simply reuses the existing code without changes. Addition of new parameter implies another instantiation of *Parameter* class in design B, but a new class is to be defined in design A. A new data structure for the parameter needs to be defined only when its particular implementation in the device is different from other parameters. Thus design B reuses the *Parameter* object and the condition method across a family of products. In design A, such changes typically require either addition of a new class or a modification of an existing method.

Time and memory requirements. The benefits of design B does come at a small penalty in execution time. Because of the number of interacting objects involved in design B, the number of method calls for setting a parameter is four while the same is achieved in two calls in design A. Further, accessing database or configuration file can significantly slowdown verification. Pre-fetching constraints from the database at the beginning of execution offsets this to some extent. Pre-compiled configuration files further optimizes time in design B.

No significant change in memory requirements is seen when the number of constraints is less. This is because the space required for duplication of constraints in multiple *condition()* methods in design A is offset by the implementation of more classes in design B. However, when the constraint set is large and they are stored in a database, implementation of design B is more compact than that of design A.

In the specific case of Programmer, time and memory are not as critical as in the implantable PG. Thus, even a simple implementation of design B is preferable to design A, because of the implications for maintenance and testing discussed earlier. However for constraint verification in real-time systems, improvements in the form of pre-compiled constraint configuration files and code optimizations may be needed.

6. Conclusion

In this paper, we described how an object-oriented framework might be used for verifying start-up constraints for some critical operations. The implications of this design and the tradeoffs were discussed. This is an attempt to systematically record the experience gained in developing safety-critical systems. Considerable work still needs to be done to collect, classify and organize such information from software design engineers working on high-assurance systems in various areas.

7. References

- [1] K. A. Ellenbogen, Editor, *Cardiac Pacing*, Second Edition, Practical Cardiac Diagnosis Series, Blackwell Science, Cambridge, MA, 1996.
- [2] E. Gamma, R. Helm, R. Johnson, and J Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, MA, 1995.
- [3] D. Lea, *Concurrent Programming in Java*, Addison-Wesley, Reading, MA, 1996.
- [4] R. Mojdehbakhsh, W. T. Tsai, S. Kirani, and L. Elliott, "Retrofitting Software Safety in an Implantable Medical Device", *IEEE Software*, Jan 1994, pp. 41-50.
- [5] A. K. Onoma, W. T. Tsai, M. Poonawala, and H. Sukanuma, "Regression Testing in an Industrial Environment", *Communications of the ACM*, May 1998.
- [6] M. Poonawala, S. Subramanian, W. T. Tsai, R. Vishnuvajjala, R. Mojdehbakhsh, and L. Elliott, "Testing Safety-Critical Systems -- A Reuse-Oriented Approach", *Proc. of the 9th Int'l Conference on SEKE*, Knowledge Systems Institute, Skokie, IL, 1997, pp. 271-278.
- [7] A. Selic and P. T. Ward, "The Challenges of Real-Time Software Design", *Embedded Systems Programming*, Vol. 9, No. 11, October 1996, pp. 66-82.
- [8] S. Subramanian and W. T. Tsai, "Backup Pattern: Designing Redundancy in Object-Oriented Software", in *Pattern Languages of Program Design 2*, edited by J. M. Vlissides, J. O. Coplien and N. L. Kerth, Addison Wesley, Reading, MA, 1996, pp. 207-225.
- [9] W. T. Tsai, R. Mojdehbakhsh, and S. Rayadurgam, "Experience in Capturing Requirements for Safety-Critical Medical Devices in an Industrial Environment", *Proc. of IEEE High-Assurance System Engineering*, 1997, pp. 32-36.
- [10] W. T. Tsai, R. Mojdehbakhsh, and F. Zhu, "Ensuring System and Software Reliability in Safety-Critical Applications", *Proc. of 1st IEEE Application-Specific Systems and Software Engineering Technology*, 1998.
- [11] W. T. Tsai, R. Mojdehbakhsh, and S. Rayadurgam, "Capturing Safety-Critical Medical Requirements", *IEEE Computer*, Vol. 31, No. 4, April 1998, pp. 40-42.