Reactive Cross-Check[1]

Dietmar Schütz
Siemens AG
ZT SE 1
dietmar.schuetz@mchp.siemens.de

The *Reactive Cross-Check* architectural pattern applies to interactive software systems which enable a human operator to supervise and control a potentially harmful real process and therefore must guarantee the correctness of all status information displayed. By cross-checking computed results against duplicated components, the Reactive Cross-Check provides safety through an "almost certain" detection of falsification of collected and visualized data.

Context

An interactive application acts as a man-machine-interface on top of a potentially harmful process. Inherent in the process is a potential danger to expensive equipment, the environment, or human life. The set of commands to operate the process is (or can be) divided into two disjoint subsets: *normal* operations yield high safety by using build-in constraints, whereas *exceptional* operations are less restrictive and therefore potentially unsafe.

Example

Signal boxes (US: switch towers) for railroad traffic are very safe, especially when following the German Railway standards. Under normal operation, they never ever set a signal to "green" until the line ahead is clear. They do not permit entrance to a section which is already occupied by another train, or blocked by a maintenance gang.

Special circumstances require to bypass this high safety level however, such as shunting (US: switching) or to tow a broken engine. In this case, the responsibility for safety is passed from the switch tower to a human operator. When entering the corresponding, safety critical command, the human operator has to rely on the *correctness* of status information displayed by the operating console. This accuracy is essential for proper judgement of the situation and input to additional measures, such as instructing the engine driver or warning the gang. Under such circumstances, every "fallen" bit can lead to disastrous consequences.

Problem

The feasibility to launch safety critical commands without using the build-in safety constraints causes a shift of responsibility from the control system via it's display to the operator. Although a falsified display, originated for example by a transient memory failure, is very unlikely, the probability of an error and the resulting risk is too high to be taken. How can the risk be reduced to an acceptable level?

The following *forces* influence the solution:

- There are many different sources for potential errors, such as hardware or network defects as well as occasional memory or transmission failures. These error sources are too various to be treated separately. The solution should be general, covering the *whole* path of information from its initial acquisition to its final visualization, and ideally work even in cases not thought about.
- Explicit polling of process data usually initiates a *general update*; fetching single attribute values is rarely supported. In addition, requesting process data from external sources (field hardware etc.) should not be done frequently due to small communication bandwidth.
- Since the remaining risk cannot be eliminated completely, the probability of an undetected error should be scaleable.

Solution

If you can't take the risk, reduce the probability. Therefore, take the following actions to achieve an “almost certain” detection of falsifications.

Build a semantically equal ”shadow” system which acts as a reference for comparisons. To this end, duplicate/replicate all involved computers where data is stored, evaluated or displayed. Both system parts (original as well as reference) are fed with data from one save source, but are strictly separated from each other regarding the flow of process data and information derived from it.

In *reaction* to a safety critical command entered by the human operator, run a cross-check in advance: instruct the reference system to display the same view as the operating one, and compare the views by extracting the corresponding image data from the video buffers and calculating a checksum on them. Those checksums should be the same.

A checksum mismatch indicates a falsification of data, and the system state must be considered to be unsafe. In conclusion,

⇒ Abort the initiated command.

⇒ Disable all other safety critical commands.

⇒ Take measures to achieve a safe system state if necessary.

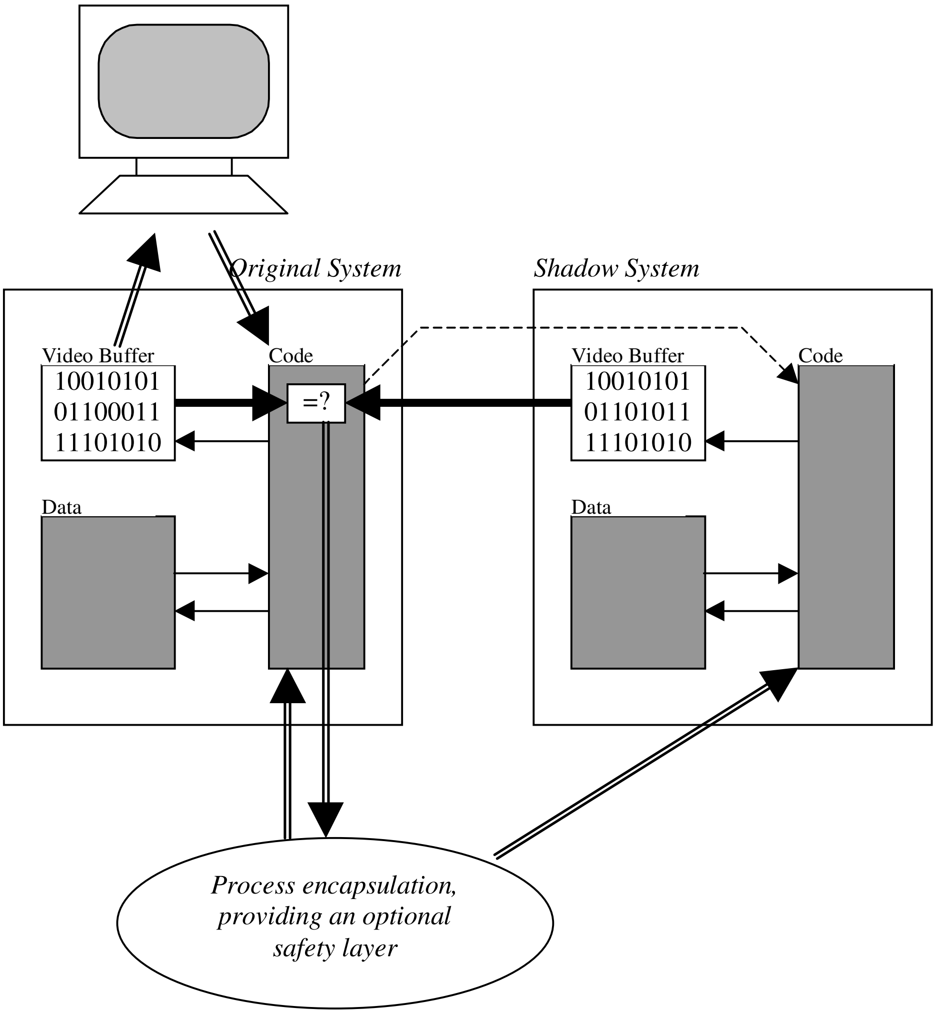
⇒ Initiate a general update to retrieve the correct data (process status and perhaps software) from a *safe* source and re-evaluate all depending information.

Structure

The *process encapsulation* provides the interface to the underlying process. It provides two sets of commands to affect the process, one with normal commands where safety purposes are handled by this component, and some additional ”special purpose” commands with no safety guarantees. This process notifies the original system as well as the shadow system in case of status changes of the underlying process.

Both the *original system* and the *shadow system* evaluate process event and keep track of the process status. They use this data to generate a display image which is stored within the video buffer.

The *original systems* responds to user commands. If these commands are not supported by the safety layer of the process encapsulation, it instructs the shadow system to display the same view, extracts the image data from the video buffers, and checks them to be identical before performing the command.



Implementation The implementation of Reactive Cross-Check needs four steps which exceed the scope a usual control system.

1 *Implement the access to the video buffer.* Since the memory on the video card is owned by the system, special access methods are needed. When extracting the image from the video buffer, don’t forget to hide the mouse cursor!

2 *Implement the checksum procedure.* The best results are achieved with the cyclic redundancy check (CRC) and generator polynoms which represent primes. The CRC method interprets the complete input as a (binary) representation of a huge number and computes the checksum as the remainder of this number and the given ”generator” value. This value, usually a prime number, limits the accurary of the checksum.

3 *Synchronization of original and shadow system.* Add a control mechanism to the evaluation of process events in order to provide a stepwise execution. When a safety critical command is initiated by the user, the original system suspends the handling of incoming process events. The sequence number of the last evaluated event is passed to the shadow system, which closes up an pauses itself. If the shadow system is ahead in execution progress, it returns its last event number back to the original. The original advances if necessary until the ”rendezvous” is made.

The following pseudo-code for this rendezvous interacts with an event processors which provides an interface for controlled, stepwise execution:
`pause()` stops processing of events; `getLastSeqNr()` retrieves the sequence number of the last completed event; `stepTo()` proceeds event processing in single steps to the give sequence number; and `continue()` resumes the execution.

Original system, method `runCrossCheck`:

```
Bool runCrossCheck() {
    theEventProcessor->pause();  // stop processing of events
    SeqNr myProgress = theEventProcessor->getLastSeqNr();
    ImageChecksum shadowImageCS = 0;
    SeqNr shadowProgess =
        theShadowSys()->syncAfter( myProgress, shadowImageCS );
    theEventProcessor->stepTo( shadowProgress );
    // Rendezvous is made, calculate checksum and compare
    ImageChecksum imageCS = theVideoBuffer->getChecksum();
    theEventProcessor->continue(); // resume event processing
    return imageCS == shadowImageCS
    }
```

Shadow system, interface method `syncAfter`:

```
SeqNr syncAfter( SeqNr orgProgress,
                 /*out*/ ImageChecksum& imageCS) {
    theEventProcessor->pause();  // stop event processing
    theEventProcessor->stepTo( orgProgress );
    SeqNr hSeqNr = theEventProcessor->getLastSeqNr();
    imageCS = theVideoBuffer->getChecksum();
    theEventProcessor->continue();  // resume event processing
    return hSeqNr;
    }
```

This example code does neither contain error recovery nor time constraints and proper timeout handling.

4 *Checksum comparison.* The simple approach would be to compare the checksums within the original system (see variants), but this causes a safety leak, containing the comparison code and the command transmission. Therefore, the comparison should be done by the process encapsulation hardware in such a way that after synchronization, both systems send as well the command code and the checksum, which must match to finally execute the command.

Known Uses

VICOS OC 111 WS, **BPS 9000** and **SIMIS-W** are Siemens switch tower systems with corresponding operation consoles. They are using the *Cross-Check* patterns to fulfill the safety requirements given by the German Railway (Deutsche Bahn, DB) standard "Mü8004", which is known as outstandingly rigorous.

Variants The Reactive Cross-Check pattern can also be applied successfully to control system software in other domains. There are two common cases:

No basic safety layer. The underlying process is not encapsulated by safety layer as given by the switch tower example. Under this circumstances, the control software must implement the checksum comparison itself and the separation as well as different treatment of the commands. Another useful option is to run a Reactive Cross-Check in advance of *all* commands.

Autonomous reactions. The control system autonomously reacts to process events by initiating automatic actions. In such case, no display or video buffer is involved. Instead, the corresponding status "image" of the affected process elements is subject to the comparison.

Consequences Before addressing concrete benefits and liabilities of this pattern, some general notes on "safety by comparison" which are to imported to be left unsaid.

There is no (nor ever will be) 100% safety. Whenever comparing two things which should be identical, there is a (very low) chance that both parts have been falsified independently in a matter that the error hide each other.

Comparison cannot detect errors which are caused by buggy software. Because both system part run the identical code, software errors will occur on both sides simultaneously, and the resulting images will show no difference.

The Reactive Cross-Check pattern provides the following **benefits**:

"Almost certain" detection of falsification. Errors from all possible sources are detected by focussing on the final image and comparing it with a reference which should look the same.

Always on the safe side. A complete chain of worst case assumptions prevents the execution of safety-critical actions not guarded by the basic safety layer as soon as even the smallest doubt arises.

Scalability. The probability of an undetected error is directly proportional to the accuracy of the checksum derived from the video buffer content. For example, a 64-bit checksum, calculated using the cyclic redundancy check (CRC) method with a suitable generator polynomial results in a probability of 10^{-19} for not detecting an error. Multiply this with the probability of any error to occur to calculate the remaining risk.

The Reactive Cross-Check pattern also suffers from the following **liabilities**:

Possibly doubles hardware costs. Implementing this pattern can almost double the amount of necessary hardware. If the original system only consisted of one operating terminal, , everything except the monitor needs to be replicated. In an environment with a "standby" system, most of the hardware is already there. Both computers only need to be equipped with an additional video card for reference purposes. In a multi-terminal environment, there is usually a spare computer which can be used as reference for comparison with no additional hardware costs.

Slows down performance. Using this pattern decreases the overall performance of the system, because it requires synchronization of original and shadow system. The comparison can only be successful if both are in the same state regarding the processed events.

Worst case overdose. In a chain of worst case assumptions, a single weak link, for example a non-writable memory bit, can prevent the system from ever reaching a safe state. In addition, it increases the system load by initialing loops of general updates.

Expensive recovery. After detecting an error, regaining a safe system state puts additional load on the system, because the data has to be retrieved from a safe source. Usually, this enforces a general update as well as reloading software parts like display routines and color maps.

Complex views. To make this pattern work, for every dangerous command there must exist at least one view which contains the all of the affected process elements in required detail. Usually, this can not be achieved by a zoomable and scrollable "big picture". Therefore, most implementations use a sufficient set of fixed size views, which causes many process element to occur in many different views. Furthermore, there must be a mapping from process elements offering dangerous commands to appropriate views.

Stale errors tend to fade. The longer a data item remains unchanged in the system, the bigger is the probability for a second, independent error to occur which prevents the detection of both errors by unfortunately making both checksums the same. Reactive Cross-Check does not include elapsed time and therefore can only detect "fresh" falsifications with the desired reliability. The time problem is addressed by the "Proactive Cross-Check" pattern, which prevents errors from getting stale.

See Also

The "Master-Slave" pattern in [POSA96] lists a variant "Master-Slave for fault tolerance", which provides the structural duplication used within Cross-Check.

A further refinement of this variant is described in the "Backup" pattern by Satish Subramanian and Wei-Tek Tsai, published in [PLoPD2].

The "Proactive Cross-Check" deals with problems which are caused by long term storing of data and are not covered by the Reactive Cross-Check.

References

[POSA] Buschmann, Meunier, Rohnert, Stal, Sommerlad;
Pattern-Oriented Software Architecture, A System of Patterns;
Wiley 1996

[PLoPD2] Vlissides, Coplien, Kerth (Editors);
Pattern Languages of Program Design, Vol. 2;
Addison Wesley 1995