Fault Management Driven Design with Safety and Security Requirements

Miroslav Sveda
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
sveda@fit.vutbr.cz

Abstract— **This paper exemplifies principles of embedded system design that props safety and security using operational errors management in frame of a dedicated Computer-Based System architecture. After reviewing basic principles of Cyber-Physical Systems as a novel slant (or marker?) to modeling and design in this domain, attention is focused on a real-world solution of a safety and security critical embedded system application offering genuine demonstration of that approach. The contribution stresses those features that distinguish the real project from a demonstration case study.**

Keywords- safety; security; operational error; fault management; embedded system design

I. INTRODUCTION

The integration of physical systems and processes with networked computing has led to the emergence of a new generation of engineered systems: Cyber-Physical Systems (CPSs) [1]. Such systems use computations and communication deeply embedded in and interacting with physical processes to add new capabilities to physical systems. Because computer-augmented devices are everywhere, they are a huge source of economic leverage. Embedded computers allow designers to add capabilities to physical systems that they could not feasibly add in any other way. By merging computing and communication with physical processes and mediating the way we interact with the physical world, CPS bring many benefits: they make systems safer and more efficient; they reduce the cost of building and operating these systems; and they allow individual machines to work together to form complex systems that provide new capabilities. By merging computing and communication with physical processes and mediating the way we interact with the physical world, CPS bring many benefits: they make systems safer and more efficient, they reduce the cost of building and operating these systems, and they allow individual machines to work together to form complex systems that provide new capabilities [1].

CPS domain paradigms [2] suggest considering both application requirements, namely time constraints defined by physical processes of the system environment, and implementation aspects, namely computation and communication capacity constraints, from the beginning of system design. The design of a CPS application should consider namely functionality and dependability measures [3].

This paper deals with design of a petrol dispenser control system as a CPS fitting special requirements not only on functionality, but also on safety and security. The key issue appears fault management in this case. Many current design methods focus on elimination of design errors, see e.g. [4], [5]. On the other hand, operational errors caused by hardware faults, varied environment, or by intentional security attacks are treated by fault tolerance and fault avoidance safety techniques or by security techniques usually during implementation [5]. This paper presents principles addressing operational errors management from the beginning through all phases of design cycle.

While the next two sections briefly introduce the Asynchronous Specification Language (ASL) [6], which is deployed for simple behavioral descriptions of the application, and dependability concepts applied, the rest of paper presents the approach to operational errors management with the design of a fuel dispenser control device.

II. SPECIFICATION LANGUAGE

A formal specification asserts that a description has precise and unambiguous semantics. The language of specification should fit purposes of specification and be appropriate for a description of the system. For structured behavioral specifications of reactive systems, process algebra CSP, temporal logic LTL and related transition systems in frame of the model checker SPIN [7] and the prover PVS [8] appear as most common tools in the domain of small embedded applications. For real-time systems, e.g. model checker UPPAAL [9] and related timed automata can be used. Unfortunately, none of these or similar generally available tools is equipped, according to our knowledge, with a simple-enough specification language that fits requirements on modeling distributed, asynchronous, multi-clocked systems implemented e.g. by multiple loosely interconnected and communicating microcontrollers in frame of a real-time application. That was the reason why we developed ASL.

The specification language ASL employs distributed sequential processes with message passing. The real-time operational semantics of the language stems from the event-count model of local time, which represents a concept of physical timing stemming from some periodic physical oscillation whose frequency fits measurements of the duration of local process actions. Timing semantics can be derived from logical time, which is a partial ordering of events in the system, and from a physical generator of periodic events, which implements a real-time clock. An event-count, E, counts the number of a specific type of events that have occurred during execution. Each event occurrence invokes the implicit operation ADVANCE(E): E:=E+1. The explicitly callable operation AWAIT(E, s) suspends the calling process until the value of E is at least s. The call AWAIT(E, s) can reset the current value of E, enabling relative counting. An event-count monitors either a prescribed type of asynchronous external events or periodic internal events that an internal timer circuit implements as local-time clock ticks. The following primitives relate to process specification, timing, communication, and control.

```
process_name(is: list_of_s_inputs; os: list_of_s_outputs;
      ic: list_of_m_inputs; oc:list_of_m_outputs):
... endprocess;

wait(_, timeout); wait(event, _); wait(event, timeout, test);
send(message, destination);
loop ... [... when <cond> action ... exit;]* ... endloop;
```

Each of asynchronous processes can be equipped by its individually timed local clock, can receive messages through input buffer, and can send messages to other, directly or indirectly addressable processes. Process header contains in parentheses lists labeled by is, os, ic, that act as the interface with the process' environment. The language distinguishes between signal inputs or outputs, which denote communication events signaling their occurrence, and message inputs or outputs as typed asynchronous channels between processes. Those signals and messages provide inter-process synchronization and communication, whose operations are driven by the statements wait(_, timeout), wait(event, _), wait(event, timeout, test), and send(message, destination).

The primitive wait(_, timeout) suspends a process for the interval defined by the value timeout. Operational semantics can be obtained through the event-count abstraction introduced above: in this case, an event is every tick of the local clock, so the related operation is AWAIT(local_ticks, timeout_value). For the primitive wait(event, _), which suspends a process until the specified event (external signal or message) appears, the model operation is AWAIT(event_type, 1). The semantics of the combined statement wait(event, timeout, test) requires two event-counts: the first anticipates the specified event and the second, with a lower priority, monitors the local clock. The reason of process activation can be checked through the value of the logical variable test: when the value is true, the event occurred within the interval timeout.

The primitive send(message, destination) implements asynchronous communication with non-blocking semantics. To respect different local clocks, a special clocking that is common for the source and the destination controls the information transfer. However, the nodes communicate asynchronously by message passing through an input buffer at the destination. The input of a message induces the event for the related operation AWAIT(message, 1). If any synchronization is required, it must be described explicitly using wait statements.

The control structure primitives loop ... endloop delimit an indefinite cycle, which is exited upon a true result of testing the condition following the primitive when. Consequently, the statements, which occur between the action and exit primitives and which follow the endloop primitive, are executed. This structured statement enables to extend the language with additional control structures by simple macro-like text replacements such as

```
if <cond> then <s1> else <s2> fi;
          ~ loop when <cond> action <s1> exit;
            <s2> when true exit; endloop;

timeloop(timeinterval) ... endloop;
          ~ loop ... wait(_, interval); endloop;
```

Actually, the control structure timeloop(timeinterval) ... endloop specifies an isochronous loop, which is periodically initiated whenever the timeinterval expires and which can be exited like the indefinite cycle. The operation AWAIT(local_ticks, timeinterval_value) defines the exact semantics of timing these initiations.

The associated rapid prototyping, which makes ASL specifications executable, arises from attribute grammar and Prolog deployment. Any Prolog interpreter can drive expansion of an ASL specification into the related executable code. This expansion is based on an attribute grammar specifying both syntax and static semantics by a definite clause grammar and Prolog rules. It provides a simple language translator prototype, which tackles the ASL as the input language, and a target executable language as the output language. The resulted prototyping technique uses interconnected node prototype boards with microprocessors equipped with a simple real-time operating system kernel. While the timing and communication primitives are mapped onto relevant real-time executive services and communication services of the operating system kernel, the rest of ASL specification is prototyped by the executable code generated with the help of the Prolog translator prototype presented in more detail by [6].

III. DEPENDABILITY

Dependability [10] is that property of a system that allows reliance to be justifiably placed on the service it delivers. A failure occurs when the delivered service deviates from the specified service. Dependability measures consist namely of reliability, availability, security, safety and survivability. Availability is the ability to deliver shared service under given conditions for a given time, which means namely elimination of denial-of-service vulnerabilities. Security is the ability to deliver service under given conditions without unauthorized disclosure or alteration of sensitive information. It includes privacy as assurances about disclosure and authenticity of senders and recipients. Security attributes add requirements to detect and avoid intentional faults. Safety is the ability to deliver service under given conditions with no catastrophic affects. Safety attributes add requirements to detect and avoid catastrophic failures.

A failure occurs when the delivered service deviates from the specified service. The failure occurred because the system was erroneous: an error is that part of the system state which is liable to lead to failure. The cause of an error is a fault. Failures can be classified according to consequences upon the environment of the system. While for benign failures the consequences are of the same order of magnitude (e.g. cost) as those of the service delivered in the absence of failure, for malign or catastrophic failures the consequences are not comparable.

A fail-safe system attempts to limit the amount of damage caused by a failure. No attempt is made to satisfy the functional specifications except where necessary to ensure safety. A mishap is an unplanned event (e.g. failure or deliberate violation of maintenance procedures) or series of events that results in damage to or loss of property or equipment. A hazard is a set of conditions within a state from which there is a path to a mishap.

A fail-stop system never performs an erroneous state transformation due to a fault. Instead, the system halts and its state is irretrievably lost. The fail stop model, originally developed for theoretical purposes, appears as a simple and useful conception supporting the implementation of some kinds of fail-safe systems. Since any real solution can only approximate the fail-stop behavior and, moreover, the halted system offers no services for its environment, some fault-avoidance techniques must support all such implementations.

Obviously, design of any safe system requires deploying security to avoid intentional catastrophic failures. And vice versa, system's security can be attacked using a safety flaw. The greater the assurance, the greater the confidence that a security system will protect against threads, with an acceptable level of risk.

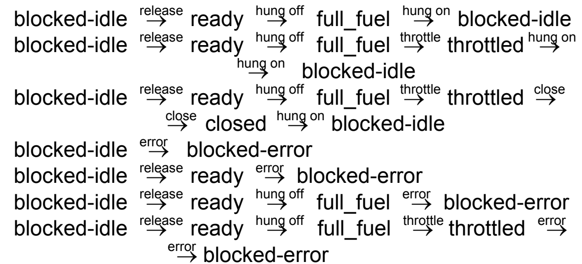
IV. APPLICATION

The application concerns petrol dispenser with an electronic counter/controller. The application appears as (1) safety critical from the point of view of danger of explosion in case of uncontrolled petrol issue and (2) security critical from the point of view of danger of loss of money in case of unregistered issue, see also [11], [12], [13].

A. State-based System Description

A dispenser control system communicates with its environment through two classes of I/O variables. The first class describes an interface with volume meter (I), pump motor (O), and main and by-pass valves (O) that enable full or throttled issue. The timing for this class is defined by flow velocity and measurement precision requirements. Second class of I/O-variables models human interface that must respect timing constants of human-physiology. This class contains release signal, unhooked nozzle detection, and product's unit prices as inputs; as for outputs, volume and price displays belong to this class.

The behavior of the higher level component can be described by the following state sequences of a finite-state automaton with states blocked-idle, ready, full fuel, throttled and closed, and with inputs release, (nozzle) hung on/off, close (the preset or maximal displayable volume achieved), throttled (to slow down the flow to enable exact dosage) and error:



The states full_fuel and throttled appear to be hazardous from the viewpoint of unchecked flow because the motor is on and the liquid is under pressure -- the only nozzle valve controls an issue in this case. Also, the state ready tends to be hazardous: when the nozzle is unhooked, the system transfers to the state full_fuel with flow enabled. Hence, the accepted fail-stop conception necessitates the detected error management in the form of transition to the state blocked-error. To initiate such a transition for flow blocking, the error detection in the hazardous states is necessary. On the other hand, the state blocked-idle is safe because the input signal release can be masked out by the system that, when some failure is detected, performs the internal transition from blocked-idle to blocked-error.

Of course, an equivalent of the above state sequences can be derived more rigorously as the Kripke-style semantics of Linear Temporal Logic (LTL) formulae specifying a related transition system, see e.g. [7], [8]. But such a formal approach was refused by cooperating development engineers from industry.

B. Incremental Measurement for Flow Control

The volume measurement and flow control represent the main functions of the hazardous states. The next applied application pattern, incremental measurement, means the recognition and counting of elementary volumes represented by rectangular impulses, which are generated by a photoelectric pulse generator. The maximal frequency of impulses and a pattern for their recognition depend on electro-magnetic interference characteristics. The lower-level application patterns are in this case a noise-tolerant impulse detector and a checking reversible counter. The first one represents a clock-timed impulse-recognition automaton that implements the periodic sampling of its input with values 0 and 1. This automaton with b states recognizes an impulse after b/2 (b>=4) samples with the value 1 followed by b/2 samples with the value 0, possibly interleaved by induced error values, see an example timed-state sequence:

$$(0, q_1) \xrightarrow{inp=0} \dots \xrightarrow{inp=0} (i, q_1) \xrightarrow{inp=1} (i+1, q_2) \xrightarrow{inp=0} \dots \xrightarrow{inp=0} (j, q_2) \dots$$
$$\dots \xrightarrow{inp=1} (k, q_{b/2+1}) \xrightarrow{inp=1} \dots$$
$$\dots \xrightarrow{inp=1} (m, q_{b-1}) \xrightarrow{inp=0} (m+1, q_b) \xrightarrow{inp=1} \dots \xrightarrow{inp=1} (n, q_b) \xrightarrow{inp=0/IMP} (n+1, q_1)$$

i, j, k, m, n are integers representing discrete time instances in increasing order.

For the sake of fault-detection requirements, the incremental detector and transfer path are doubled. Consequently, the second, identical noise-tolerant impulse detector appears necessary.

The subsequent lower-level application pattern used provides a checking reversible counter, which starts with the value (h + l)/2 and increments or decrements that value according to the impulse detected outputs from the first or the second recognition automaton. Overflow or underflow of the pre-set values of h or l indicates an error. Another counter that counts the recognized impulses from one of the recognition automata maintains the whole measured volume. The output of the letter automaton refines to two displays with local memories not only for the reason of robustness (they can be compared) but also for functional requirements (double-face stand). To guarantee the overall fault detection capability of the device, it is necessary also to consider checking the counter. This task can be maintained by an I/O watchdog application pattern that can compare input impulses from the photoelectric pulse generator and the changes of the total value; evidently, the appropriate automaton provides again reversible counting.

Similarly like in the previous subsection, the more formal approach can be based on some real-time temporal logic, e.g. TLTL, or a more simple temporal logic equivalent to counting automata, [9]. But again, such a formal approach was rejected by cooperating development engineers from industry.

C. Behavioral Specification

The demonstration of logical structure description employs a fast process simulating both of the two impulse-recognition automata together with the reversible counter. The detection process sends a message about a detected impulse to the slower meter process, which sends a fuel-volume message to the display process.

A high-level process simulates the previously discussed behavior of the dispenser. For that reason, a communication between the dispenser-control process and the above-described lower-level processes must proceed. Usually, the design progresses top-down. Hence, the primarily designed fuel-stand process reads the input variable fuel-volume. A next refinement replaces the simple reading by the communication with the meter process from the lower level. Similarly, the write commands to block output expand to a communications with the blocking process.

```
process detection (s: hang_off, hang_on; o: meter):
loop   q0 := 1; q1 := 1; count := (h+l)/2; wait(hang_off, _);
  timeloop(sample_interval)
    read(in0,input0); read(in1,input1);
    if q0 <= n/2 then begin if in0 = 1 then q0 := q0 + 1 end
    else if in0 = 0 then q0 := q0 + 1;
    if q1 <= n/2 then begin if in1 = 1 then q1 := q1 + 1 end
    else if in1 = 0 then q1 := q1 + 1;
    if q0 >= n then begin q0 := 1; count := count - 1;
                  send(impulse,meter) end;
    if q1 >= n then begin q1 := 1; count := count + 1 end;
   when l > count or count > h action write(true,block) exit;
  endloop;
  wait(hang_on, _);
endloop;
endprocess;

process meter (s: hang_off; i: impulse; o: display):
loop   vol := 0;
  loop   read(position,nozzle);
  if position = hang_on then begin vol:=0; wait(hang_off, _);
                                 send(vol,display) end;
      wait(impulse, _); vol := vol + 1;
      when vol > maxvol action write(true,block) exit;
      send(vol,display);
  endloop;
endloop; endprocess;
process display (i: vol):
loop   write(vol, display1); write(vol, display2);
       wait(vol, update_interval, test); endloop;
endprocess;
```

D. System Structure Refinement

The reviewed design example complies with such decisions as incremental measuring, periodic sampling of impulses, doubling the incremental detector and transfer path, and choosing the nozzle position for synchronization. Evidently, these design patterns support the considered fail-stop model.

Next patterns have to bring suitable solution of the dispenser control system for achieving broader applicability. Dispenser is a ranged product, so the minimal production costs are required. This requirement leads to a multi-purpose device for petrol, octane mixture, petrol and additive mixture, or high-speed diesel-oil issue, for the attendant station or for the self-service station with cashier or with debit or credit automaton or slot machine. One of the functions enables to preset the fuel (centrally by the cashier or locally on the stand) in volume or cash with the automatic termination of the dose.

The physical design of the system is based on a distributed architecture with optionally two or three simple microcontrollers (if the preset unit has been installed) as depicted on Figure 1. and Figure 2., see also the following page. While the management system, if present, participates in the data communication architecture, the debit or credit automaton or slot machine observes only volume impulses and rules release, throttle, and close signals, completing product issue independently.

The microcontrollers interact so the auxiliary (A) and main (M) processors are configured front-end/back-end with regard to impulse pipelining while the main processor, M, preset unit processor, P, and/or management system processor form a master/slave configuration for the transfer of the preset or completed fuel volume. The main microcontroller, M, implements the volume meter, dose/cash counter, main display service, and stand driver, including fuel control. The auxiliary microcontroller, A, pre-processes the volume impulses of both the possible liquids and implements testing and checking functions. The preset unit processor, P, serves both keyboard and local display and calculates a volume equivalent if the pre-setting is in cash. In between processors A and M there is a watch-dog, designed for guarding the equivalent main display increments with respect to the primary impulses. All three processors share access to the actual unit prices and mix-ratio through a multiplex driven by the main processor.

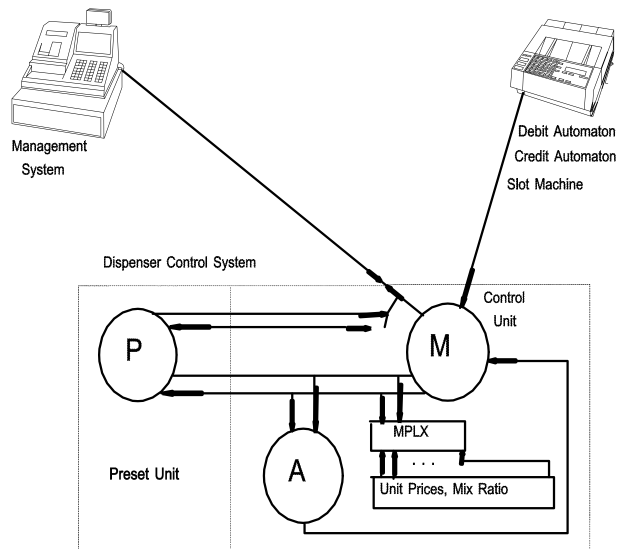


Figure 1. Configuration

Remaining input is nozzle position. Outputs control pump motor, main/throttle valves and signal lights drivers. In the state blocked-idle with both valves closed and the motor off, the red light only is on; in the state ready with both valves closed and the motor off, the green light only is on; in the states full fuel (both valves opened, the motor on), throttled (the main valve closed, the throttle valve opened, the motor on), and closed (both valves closed, motor off) both lights are off; at last, the state blocked-error (both valves closed, motor off) is signaled by both lights on.

The detailed system logical design respects hard-real-time limits for impulse inputs and a response-time limit related to the preset fuel-dose completion. Processes located to the main and auxiliary processors are implemented in foreground/ background format so that the time-critical sequences are triggered by interrupts generated by local timers. The software of the preset unit processor includes an isochronous loop for keyboard debounce. The presence of the preset unit and/or central cashier system has to be transparent for the rest of the control system software. The framework includes a corresponding data-communication protocol that provides also optional installation of the preset unit, and/or a management system, see the following algorithmic specifications of the processes M and P.

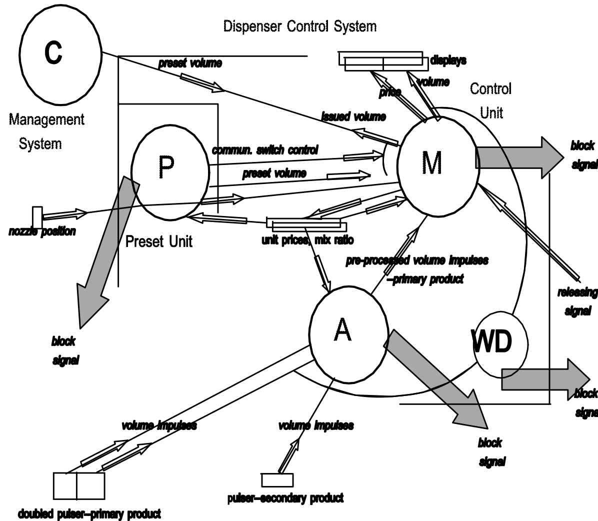


Figure 2. Communication structure

```
process M (i: preset, confirmation; o: request, total):
transaction_m(peer, request, response, result)
    result := false; send(request, peer);
    wait(response, m_timeout, test); if test then result := true;
end;
loop
    dispenser_status := blocked-idle;
;dispenser_status = blocked-idle
    write(false, unblock);
    timeloop(repeat_interval)
  read(connected, C_ready)
  if connected then begin read(position, nozzle);
     if position = hang_on
            then begin
                        transaction(C,request,preset,success);
            when success action exit; end;
                  end
            else begin read(liberate, release);
 when liberate action preset:=(max_vol, mix_free) exit;   end;
endloop;
    dispenser_status := ready;
;dispenser_status = ready
    if mix_ratio = mix_free
 then timeloop(repeat_interval)
  read(position, nozzle); when position=hang_off action exit;
      transaction_m(P, request, preset, success);
      when success action exit;
endloop;
    dispenser_status := full_fuel;
;dispenser_status = full_fuel
    write(true, unblock);
            .
            .
    dispenser_status := blocked-idle;
;dispenser_status = blocked-idle
    write(false,unblock);
    timeloop(repeat_interval)
  read(connected, C_ready); when not connected action exit;
  transaction_m(C, total, confirmation, success);
  when success action exit;
    endloop;
endloop
endprocess;
```

```
process P (i: request; o: preset):
transaction_s(peer, request, response, result)
    result := false; wait(request, s_timeout, test);
   if test then begin result := true;  send(response, peer); end;
end;

loop      .
;dispenser_status = ready
    write(connect_PM, switch);
    loop read(position, P_nozzle);
  when position = hang_off action exit;
  transaction_s(M,request, preset, success);
  when success action exit;
    endloop;
    write(hang_off,nozzle); write(connect_CM, switch);
;dispenser_status = full_fuel
    write(true, unblock);
endloop;
endprocess;
```

V. RESULTING FAULT MAINTENANCE CONCEPTS

After the text edit has been completed, the paper is ready for the template. Duplicate the template file by using the Save As command, and use the naming convention prescribed by your conference for the name of your paper. In this newly created file, highlight all of the contents and import your prepared text file. You are now ready to style your paper.

The application discussed appears as safety critical because of (1) danger of explosion in the case of uncontrolled petrol issue and (2) loss of money in the case of unregistered issue. The first item, excepting security management with debit or credit automaton and slot machine, is resolved without any support of the electronic counter/controller (nozzle with hydraulic shut-off, hooked nozzle mechanical blocking, and cashier administration). To prevent unregistered issue, the fail-stop conception used appraises as more acceptable the forced blocking of the dispenser with preserved actual data on displays instead of an untrustworthy issue. On the other hand, because permanent blocking or too frequently repeated blocking is inappropriate, the final implementation must employ also fault avoidance techniques. The next reason for the fault avoidance application stems from the fact that only approximated fail-stop implementation is possible.

The configurations, so far introduced stepwise, accomplish the fault management in the form of (a) hazardous state reachability control and (b) hazardous state maintenance. In all safe states (blocked-idle, closed, and blocked-error), any fuel issue is disabled by power hardware construction; in the same time, the contents of

all displays are protected against any change required by possibly erroneous control system. The system is allowed to reach hazardous states (ready, full fuel, and throttled) when the processors successfully passed the following tests: start-up checks, unit prices comparison, inter-processor communication, and all-or-nothing voting. The hazardous state maintenance includes doubled input path check for a main product, mixture ratio check for secondary product, watchdog check, and passive display test.

After power reset, all microcontrollers installed perform start-up checks, which encompass internal RAM test, ROM checksum test, and timer functional test. The dispenser can be released either by the cashier at the petrol station management system through the above depicted data communication protocol or by a release signal from attendant/cashier, debit automaton, credit automaton, or from slot machine. In the first case, a communication transaction must proceed between the dispenser's main processor and the management system's processor; similarly, a local preset leads to a communication transaction between the dispenser's main and preset processors. When the nozzle is being unhooked, all microcontrollers installed check the multiplex function together with unit price settings, which form two doubled independent inputs on dual faced calculator for both possible products. After that, all processors vote if the motor can be started and both valves opened. All processors must agree to enable the issue. If one of them votes against because one or more of the previously mentioned tests have not passed, the dispenser transfers to the state blocked-error and the issue is blocked until next reset after repair.

When the dispenser issues a product mixture with a ratio setup before the nozzle is unhooked, the above-mentioned reversible counter performs the check of main product impulses, which are doubled by doubling the impulses source and the transfer path. Also, the possible secondary product impulses are checked with the adequate main product impulses and the ratio setup, using similar reversible counter. The output information changes, represented by low-order bit position sent to volume displays, are checked by the watchdog--in this case an independent hardware reversible counter--against the main product impulses. As a result, differences bigger than a tolerated value can also result in issue blocking. The last test deals with the possibility to check all 7 segments in all positions whenever a button is pressed. An attendant can check if all display positions exhibit the figures "8". When the button is loosening, the actual output information, saved in display buffers, appears on displays. In the case of a detected error, the attendant must decide about proper maintenance.

Of cause, the above-described patterns create only skeleton carrying common fault-tolerant configurations. In short, while auxiliary hardware components maintain supply-voltage level and reset, nozzle position, and release signals filtering and timing, the software techniques, namely time redundancy or skip-frame strategy; deal with non-critical inputs and outputs.

VI. Conclusions

This paper describes a fault management example in frame of a safe and secure embedded system using dedicated architecture. After reviewing the simple specification language and dependability concepts deployed, main attention is focused on hardware architecture, software, and communication services fitting the application requirements. The petrol dispenser controller exemplifies in this case a real-world solution of a safety and security critical embedded system application. The presented paper updates and refines the approach originally introduced in [14] for another application.

The full behavioral specification discussed above was prototyped using the technique mentioned at the end of the section 2 that resulted in executable model heavily utilized for experiments during not only early design phases, but also later on when investigating variants for reuse and re-design of the application. The formal semantics of the ASL enabled also to develop a related model-checking technique, which was used, due to exponential complexity, to verify only selected parts of the specification.

The reason of developing a new specification language deals with its asynchronous, multi-clocking nature together with overall simplicity, which was required by measurement and pumping application domain developers.

Acknowledgment

The research has been supported by the Czech Ministry of Education in the frame of Research Intentions MSM 0021630528: Security-Oriented Research in Information Technology and MSM 0021630503: MIKROSYN: New Trends in Microelectronic Systems and Nanotechnologies; and by the Grant Agency of the Czech Republic through grant GACR 102/08/1429: Safety and Security of Networked Embedded System Applications.

References

[1] Krogh, B.H., E. Lee, I. Lee, A. Mok, R. Rajkumar, L.R. Sha, A.S. Vincentelli, K. Shin, J. Stankovic, J. Sztipanovits, W. Wolf, and W. Zhao, *Cyber-Physical Systems, Executive Summary*, CPS Steering Group, Washington D.C., 16pp., March 6, 2008. [http://www.nsf.gov/pubs/2008/nsf08611/nsf08611.htm]

[2] Stankovic, J.A., I. Lee, A. Mok, and R. Rajkumar, "Opportunities and obligations for physical computing systems," *IEEE Computer*, November 2005, pp.23-31.

[3] Jackson, E.K. and J. Sztipanovits, "Correct-ed through Construction: A Model-based Approach to Embedded Systems Reality," *Proceedings 13th Engineering of Computer-Based Systems*, IEEE Computer Society, Los Alamitos, CA, 2006, pp.164-173.

[4] Bowen, J. P., and M. G. Hinchey, *High-Integrity System Specification and Design,* Springer, New York, 1999.

[5] Henzinger, T.A. and J. Sifakis, "The discipline of embedded systems design," *IEEE Computer*, Vol.40, No.10, 2007, pp.36-44.

[6] Sveda, M., and R. Vrba, "Executable Specifications for Distributed Embedded Systems," *IEEE Computer*, Vol.34, No.1, 2001, pp.138-140.

[7] Holzmann, G.J., "The Model Checker Spin," *IEEE Transactions on Software Engineering*, Vol.23, No.5, 1997, pp.279-295.

[8] Owre, S., J. Rushby and N. Shankar, "PVS: A Prototype Verification System, Automated Deduction," (D. Kapur, Ed.), *Lecture Notes in Artificial Intelligence*, Springer, New York, USA, Vol.607, 1992, pp.748-752.

[9] Larsen K.G., L. Pettersson, Wang Yi, "Uppaal in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, Vol.1, No.1-2, 1997, pp.134-152.

[10] Melhart, B. and S. White, "Issues in Defining, Analyzing, Refining, and Specifying System Dependability Requirements," *Proceedings ECBS'2000*, IEEE CS, Edinburgh, Scotland, 2000, pp.334-340.

[11] Sveda, M., "Formal Specs Reuse with Embedded Systems Design -- Behavioral and Architectural Specifications in Real-Time Application Domains," *Proceedings ICONS 2007*, IEEE CS, New York, USA, 2007, pp.11-16.

[12] Sveda, M., "Application Patterns for CBS Design Reuse." *Proceedings of the IEEE Conference and Workshop ECBS'99*, Nashville, TN, USA, IEEE Computer Society Press 1999, pp.92-98.

[13] Sveda, M., "Embedded System Design: A Case Study." *Proceedings IEEE Symposium and Workshop ECBS'96*, Friedrichshafen, Germany, IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 260-267.

[14] Sveda, M., "Fault Management for Secure Embedded Systems," *Proceedings ICONS 2009*, IEEE Computer Society, New York, USA, 2009, pp.23-28.