Improving Dependability of Automation for Free Electron Laser FLASH

Bogusław Kosiela¹, Tomasz Szmac², Wojciech Cichalewski¹

¹ Technical University of Łódź,
Department of Microelectronics and Computer Science,
Al. Politechniki 11, 90-924 Łódź, POLAND
² AGH University of Science and Technology, Department of Automatics
Al. Mickiewicza 30, 30-059 Kraków, POLAND

Abstract. Free-electron laser FLASH (260-meter-long machine) [1] is a pilot facility for the forthcoming XFEL [1] (3 km) and ILC [2] (~35 km) projects. Along with growth of the experiment, service and maintenance are becoming so complex that certain degree of automation seems to be inevitable. The main purpose of the automation software is to facilitate operators with computer-aided supervision of several hardware/software subsystems. The efforts presented in this contribution concern elaboration of general framework for designing and development of automation software for the FLASH. The toolkit facilitates specification, implementation, testing, and formal verification. The ultimate goal of the framework is to systematize the way of automation software development and to improve its dependability. At present usefulness of the tools is being evaluated by testing the automation software for single RF-power station of the FLASH.

0.1 Safety-Critical Aspects of Application

The peculiarity of application implies number of requirements typical for safety-critical applications. One of the most important feature of customer-oriented facility as FLASH is machine uptime. It entails automation software liveness. Moreover, the software deals with expensive hardware installations, which are now and then serviced by technical personnel. Under this circumstances assertion of various safety properties is evident.

0.2 Insufficiencies of Previous Frameworks

Several attempts to automate certain subsystems of the FLASH have been performed at the DESY [4 7]. All of them utilized the DOOCS [3] Finite State Machine [8, 4] toolkit or Stateflow [9].

Authors’ practice reveals that successful applications of simple automation schemes are feasible but design of statemachines for larger subsystems turns out to be tedious and error-prone. The problem becomes particularly evident when specification evolves and design has to be updated. Then, even well elaborated statemachine becomes a mixture of complex expert’s knowledge and tricky...
endeavors which make it work. Both aforementioned toolkits offer merely the implementation tools. They do not facilitate stages of specification testing and verification.

1 Proposed Approach

To address requirements of application domain, several mechanisms borrowed from expert-systems field have been used. Proposed software consists of two execution engines supplied with the specification in the domain-specific language. The planner engine assembles plans to drive the subsystem automatically towards desired operation mode. The exception handler is designed to deal with possibly complex exceptional situations, which may be exposed by driven hardware. Its role is to fix known operation glitches and perform conflict resolution in the case of multiple exceptions.

2 The Architecture

Single installation of the automation software consists of two runtime automation engines and two specification files. Cooperation of the engines is realized by dedicated cooperation protocol.

\[\text{Fig. 1. Single installation of the automation software.}\]

2.1 Planner Engine

Its role is to automate routine operation procedures usually performed by the operators. It consists of specification language interpreter, state estimator, planner and plan executor. State estimator retrieves current status of supervised accelerator subsystem. Planner synthesizes a sequence of procedures bringing the
system from active state to the state satisfying specification of target operation mode. Plan executor takes care for executing a single procedure. The specification for the planner engine is comprised of constructs presented by the grammar from Fig. 2. A state space of a finite state description is represented by set of system variables (<variable>) with significantly reduced domains. Physical signals readouts are introduced to the specification by means of <observable>. Mapping between the model and hardware readouts is accomplished by definition of system variables domains. Possible model state transformations are expressed by means of atomic operations (<procedure>). Their specification consists of precondition, postcondition, reference to the executable code and estimate of execution time. Procedure is permissible only if its precondition evaluates to true. Postcondition becomes fulfilled after its successful execution. Execution time helps in estimation whether the procedure is still in progress or has presumably failed. Since every automated operation is performed on purpose, there is a way to specify possible goals of automation. For these purpose there exists a construct <op_mode>. It specifies a valuation of subset of system variables which must hold for the operation mode to be active. Specification can be augmented with definitions of formal properties of the model (<formal_prop>). The only usage scenario of the planner engine is to configure target mode and let the software bring the subsystem there. This process executes in cycles. Every cycle the state estimator guesses the status of supervised device, planner finds the sequence of atomic procedures driving the system into target operation mode and plan executor performs first procedure from the plan. After reaching the target operation mode, planner engine restricts itself to monitoring. In the case of single procedure failure several scenarios depending on plan executor setup may happen. At present there are two setups possible. First repeats failed procedure while the second tries to find and execute alternative procedure.

2.2 Handler of Exceptional Events

Exception handler recognizes operation glitches and if possible executes appropriate remedy procedures. If exception cannot be dealt with automatically, it stops the automation software and warns the operators. In the case of multiple exceptions it must choose the most suitable remedy procedure.

Its specification language is designed for definition of exceptional situations. They are described by means of conditions defined in terms of monitored DOOCS properties\(^3\). There are distinguished three categories of the exceptions. Permanent faults, temporary interrupts and warnings. Faults cause permanent break in machine operation. Interrupts are temporal glitches which can be automatically dealt with. Warnings provide information about possibly approaching operation problems.

Above classification was introduced to facilitate conflict resolution in the case of multiple exceptions occurrence. If a fault occurs, automation software is permanently suspended and appropriate message is sent to operators' console.

\(^3\) Corresponding grammar may be found in Fig. 2
Fig. 2. Grammar defining syntax of the specification language for both the planner and the exception handler.
Occurrence of an interrupt in case of lack of faults entails execution of suitable remedial procedure.

Conflict resolution between interrupts is based on calculation of subsumption relation. More strictly specified interrupts have precedence before more general ones. The algorithm for deciding whether one exception subsumes another utilizes two constraint solvers, The cly bounds [16] and the clypr [16]. The idea of calculating the relation is fairly simple. If one assumes two exceptions \( E_1 \) and \( E_2 \) which conditions \( \text{cnf}_1 \) and \( \text{cnf}_2 \). The algorithm reports the subsumption if there exist three valuations \( V_1, V_2, V_3 \) of variables (hardware readouts) in \( \text{cnf}_1 \) and \( \text{cnf}_2 \) meeting one of the following statements.

\[
E_1 \text{ subsumes } E_2 \text{ iff }
\begin{align*}
(cnf_1(V_1) & \land cnf_2(V_2)) \land (cnf_1(V_2) \land \neg cnf_2(V_2)) \\
& \land \neg (\neg cnf_1(V_3) \land cnf_2(V_3))
\end{align*}
\]

\[
E_2 \text{ subsumes } E_1 \text{ iff }
\begin{align*}
(cnf_1(V_1) & \land cnf_2(V_2)) \land (\neg cnf_1(V_2) \land cnf_2(V_2)) \\
& \land \neg (cnf_1(V_3) \land \neg cnf_2(V_3))
\end{align*}
\]

When above conflict resolution methods fail, the order of appearance in the specification file decides which exception is handled first.

2.3 Cooperation Scenarios

Both the runtime engines perform complementary tasks. Since they share the same hardware equipment, they must obey certain rules of cooperation. For this purpose a protocol orchestrating their collaboration has been designed. General diagram of the cooperation protocol design is presented in Fig. 3. Table 1 explains the interfaces presented in the diagram. Figures 4 and 5 present the design of the cooperation protocol in the form of Harel’s statecharts [19]. Table 2 provides descriptions of the states from the Fig. 4 and 5. Poorly designed cooperation protocol might cause automation software to hang. Therefore it had to be verified for the deadlock [18] and livelock [18'] freedom. The SPIN [14] model checker was used for this purpose. Protocol design presented in Fig. 3, 4 and 5 was modeled in the PROMELA\(^4\) language. Then the model has been checked for the existence of deadlocks and livelocks.

It turned out that all non-progressive cycles [14] found in the model did not cause starvation (livelock). They have been marked as progressive by inserting a progress labels depicted as numbered bullets in 4 and 5. After inserting the labels into the model no invalid endstates [14] (deadlocks) have been found. Besides, the model has been verified to conform to its functional requirements presented in Fig. 6.

\(^4\) It is assumed that they are interpreted according to the operational semantics described in the “Stateflow User’s Guide” [9]

\(^5\) A modeling language of the SPIN model checker
Fig. 3. General scheme of communication between the planner and the exception handler.

Table 1. Explanation of the data supplied to and interchanged between the parts of the protocol from Fig. 3

<table>
<thead>
<tr>
<th>Stimulus name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_RECOGNIZED</td>
<td>State of supervised plant fits in the state space defined by the specification</td>
</tr>
<tr>
<td>GOAL_REACHED</td>
<td>A state of the target operation mode has been reached</td>
</tr>
<tr>
<td>PLAN_SUCC</td>
<td>A path to one of the target states has been found</td>
</tr>
<tr>
<td>GOAL_AIMED</td>
<td>Target operation mode has been specified</td>
</tr>
<tr>
<td>AUTO_MODE</td>
<td>The software is permitted to supervise the plant</td>
</tr>
<tr>
<td>GLITCH</td>
<td>Exception handler reports an operation glitch</td>
</tr>
<tr>
<td>ERROR</td>
<td>Exception handler reports a permanent fault</td>
</tr>
<tr>
<td>FREEZE_PLANNER</td>
<td>Suspends planner</td>
</tr>
<tr>
<td>FREEZE_HNDLR</td>
<td>Suspends exception handler</td>
</tr>
</tbody>
</table>

Fig. 4. Design of the communication protocol for the planner.
Table 2. Explanation of state names from the Fig. 4 and 5

<table>
<thead>
<tr>
<th>State name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_AUTO</td>
<td>Planner is permitted to supervise the plant</td>
</tr>
<tr>
<td>P_FROZEN</td>
<td>Planner is suspended</td>
</tr>
<tr>
<td>GOAL_IS_AIMED</td>
<td>Target operation mode has been specified</td>
</tr>
<tr>
<td>GOAL_NOT_AIMED</td>
<td>Target operation mode is not specified</td>
</tr>
<tr>
<td>STEP_EXECUTION</td>
<td>Planner executes single step of the plan</td>
</tr>
<tr>
<td>PLANNING</td>
<td>Planning in progress</td>
</tr>
<tr>
<td>STATE_SCANNING</td>
<td>Planner performs state recognition</td>
</tr>
<tr>
<td>INCOMPLETE</td>
<td>Planner is incomplete</td>
</tr>
<tr>
<td>INC_PLANNING</td>
<td>Planning procedure failed</td>
</tr>
<tr>
<td>INC_STATE_SCANNING</td>
<td>State of the plant is unknown</td>
</tr>
<tr>
<td>E_MANUAL</td>
<td>Both engines are suspended</td>
</tr>
<tr>
<td>E_FROZEN</td>
<td>Exception handler is suspended</td>
</tr>
<tr>
<td>AUTO</td>
<td>Both engines are permitted to supervise the plant</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Exception detection in progress</td>
</tr>
<tr>
<td>REPORTING_EVENTS</td>
<td>All exceptions are being reported to the operator</td>
</tr>
<tr>
<td>PROCEDURE_EXECUTION</td>
<td>Exception handling procedure in progress</td>
</tr>
</tbody>
</table>

1. FREEZE_PLANNER → ¬FREEZE_PLANNER
2. FREEZE_HNDLR → ¬FREEZE_HNDLR
3. GLITCH → ¬P_FROZEN
4. ERROR → ¬P_FROZEN
5. MANUAL → ¬P_FROZEN
6. ¬GOAL_AIMED → ¬GOAL_NOT_AIMED ∧ E_FROZEN

Fig. 6. Properties of the cooperation protocol which prove its responsiveness and deadlock freedom.
3 Integrated Formal Verification and Testing

In this project, automated formal verification is realized by model checking [12]. The NuSMV [11] is a model checker used to verify formal properties included in the specification for the planner. Dedicated converter translates the model encoded in the specification language to the equivalent model expressed in the NuSMVs input language. Definitions of formal properties which need to be fulfilled by the model are expressed in the Computation Tree Logic (CTL) [12]. Fragmentary example of the NuSMV input specification can be seen in Fig. 7.

```plaintext
MODULE systems_state
VAR
  FORCE_MANUAL_MODE: [FALSE, TRUE];
  MODULATOR_STATUS: [LOCKED_FOR_5_MIN, ERROR, OFF, ON];
ASSIGN
  next(FORCE_MANUAL_MODE) := FORCE_MANUAL_MODE;
  next(MODULATOR_STATUS) := MODULATOR_STATUS;

MODULE switch_to_manual(st)
ASSIGN
  next(st.FORCE_MANUAL_MODE) := case
    st.FORCE_MANUAL_MODE = TRUE : FALSE;
    st.FORCE_MANUAL_MODE = FALSE : TRUE;
    esac;

MODULE switch_to_auto(st)
ASSIGN
  next(st.FORCE_MANUAL_MODE) := case
    st.FORCE_MANUAL_MODE = FALSE : TRUE;
    esac;

MODULE main
VAR
  state : process systems_state;
  proc_switch_to_manual : process switch_to_manual(state);
  proc_switch_to_auto : process switch_to_auto(state);
FAIRNESS running

SPEC EF (state.FORCE_MANUAL_MODE = FALSE & state.KLY_INTERLOCK_STATUS = ALL_GREEN & state.MODULATOR_STATUS = ON)
```

Fig. 7. Fragmentary specification for the planner automatically translated to the NuSMV input language. The model is an asynchronous statemachine. Its state is described by symbolic variables defined in the module `sys Stamforde state` and each transition is represented by corresponding module (e.g. `switch_to_manual`). Module `main` instantiates all the processes. Model execution consists in sequential execution of non-deterministically chosen processes.

4 Testing Environment

To facilitate the process of automation design, dedicated software has been implemented. The toolbox allows to simulate continuous or step-by-step execution
of the planner engine. It provides the interface to display and simulate the system state and integrates automatic formal verification. Some elements of the toolbox can be seen in the Fig. 7, 8, 9.

![Figure 8](image.png)

**Fig. 8.** The interfaces for choosing the target operation and step-by-step simulation.

5 Proof of Concept

Usability of the framework has been evaluated by implementation of supervision software for RF-power station subsystem. This installation is responsible for

![Figure 9](image.png)

**Fig. 9.** RF-power station in the "MANUAL_MODE" operation mode.

supplying cavities with energy necessary for particle acceleration. Despite the whole RF-power station is quite complex, it has simple operation scenarios. Six operation modes have been specified. They are presented in Fig. 8. The system state was described by nine system variables depicted in Fig. 9. The exception handler was supplied with the specification of the following exceptions.
Personal interlock active (personal safety, permanent fault)
RF-leakage detected (personal safety, permanent fault)
Unrecoverable modulator fault (permanent fault)
Modulator power supplier switch is off (human assistance needed)
Only RF-inhibit activated (remote restart possible)
General modulator problem (remote restart possible)
IGCT stack overheated (hardware safety, wait till temperature drops)

The software has been used for several maintenance days for driving the one
RF-power station. Both planned and exception handler proved to work correctly.

References