Improving Dependability of Automation for Free Electron Laser FLASH

Bogusław Kosęda, Tomasz Szmuc, Wojciech Cichalewski

Abstract:

Free-electron laser FLASH (260-meter-long machine) is a pilot facility for the forthcoming XFEL (3 km). 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.

Keywords: *Automation, formal methods, model checking, expert system, Prolog, FLASH.*

1. Introduction

The peculiarity of this system implies a number of requirements typical for *safety-related applications* [15]. The automation software for supervision of the laser equipment cannot break it due to its internal logical error. It also must not expose human personnel inspecting high power laser equipment to a risk of being injured. One of the most important requirements for customer-oriented facility as FLASH [1] is maximization of *machine uptime* (the time when the FLASH is utilized for the experiments). The automation software is expected to be a means for improving this factor by reducing *human error* and by providing automatic fault recovery. Under these circumstances *liveness* of the software seems to be very important too.

Several attempts to automate certain subsystems of the FLASH have been performed at the DESY[1] [3,4,5,6]. All of them utilized the DOOCS[2] [2] Finite State Machine [7,3] toolkit or Stateflow [8]. 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 endeavours, which "make it work". Both aforementioned toolkits offer merely the implementation tools. They do not facilitate stages of specification, testing and verification.

To address requirements of application domain, several mechanisms borrowed from expert-systems field have been used. Proposed software consists of two execution engines (see Fig. 1) 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 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 a dedicated cooperation protocol.

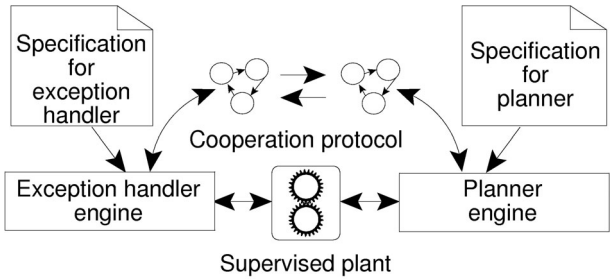

Fig. 1. Single installation of the automation software.

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 of 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 (<*qvariable*>) 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*>).

1. Deutsches Elektronen-Synchrotron in Hamburg, member of the Helmholtz Association.
2. Distributed Object Oriented Control System.

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 <*opmode*>. 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 (<*formalprop*>). 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[3] properties. 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.

```
<specification>   ::= {def <definition> ";"}
<definition>      ::= <repexceptions> | <exception> | <observable> | <procedure> | <qvariable> | <opmode> |
                      <formalprop>
<observable>      ::= observable <obsname> of type <obstype>
                      taken from <doocsaddr>
<obstype>         ::= bool | int | float | string
<condition>       ::= <relation> <junction> <relation>
<junction>        ::= & | "|"
<relation>        ::= <obsname> <operator> <number> | <obsname> <bitop> integer <operator> integer
<operator>        ::= == | != | < | > | <= | >=
<bitop>           ::= bitor | bitand | bitxor
<number>          ::= integer | float
<procname>, <obsname>, <qname>, <qval>, <opname>, <pname>, <doocsaddr>, <description>,
<message>         ::= string
<qvariable>       ::= qvariable <qname> <qdomain>
<q domain>        ::= <qvalue> {" ," <qdomain>}
<qvalue>          ::= value <qval> if <condition>
<opmode>          ::= opmode <opname> active when <state>
<state>           ::= <state> {<junction> <state> }
<state>           ::= <qrelation>
<qrelation>       ::= <qname> == <qval> | <q name> != <qval>
<procedure>       ::= procedure <proc name> description: <description> trigger: <doocsaddr>
                      allowed <condtype> <condition>  postcondition: <condition> cost: integer
<condtype>        ::= unless: | when:
<formalprop>      ::= specification <pname> <ptype> <state>
<ptype>           ::= always | never | possible
<repexceptions>   ::= report <exctype> to <doocsaddr>
<exctype>         ::= fault | warning | interrupt
<exception>       ::= <interrupt> | <fault> | <warning>
<interrupt>       ::= interrupt <description> holds if <condition> report message <message>
                      execute procedure <proc name>
<fault>           ::= fault <description> holds if <condition> report message <message>
<warning>         ::= warning <description> holds if <condition> report message <message>
```

Fig. 2. Grammar defining syntax of the specification language for both the planner and the exception handler.

3. Corresponding grammar may be found in Fig. 2.

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. 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 *clp/bounds* [12] and the *clpqr* [12]. The idea of calculating the relation is fairly simple. If one assumes two exceptions E_1 and E_2 which conditions *cnd1* and *cnd2*. The algorithm reports the subsumption if there exist three valuations V_1, V_2, V_3 of variables (hardware readouts) in cnd1 and cnd2 meeting one of the following statements.

$$E_1 \textbf{ subsumes } E_2 \textit{ iff}$$

$$(cnd1(V_1) \wedge cnd2(V_1) \wedge (cnd1(V_2) \wedge \neg cnd2(V_2)) \wedge \neg(\neg cnd1(V_3) \wedge cnd2(V_3))$$

$$E_2 \textbf{ subsumes } E_1 \textit{ iff}$$

$$(cnd1(V_1) \wedge cnd2(V_1)) \wedge (\neg cnd1(V_2) \wedge cnd2(V_2)) \wedge \neg(cnd1(V_3) \wedge \neg cnd2(V_3))$$

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 [14][4]. Table 2 provides descrip-tions 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 [13] and livelock [13] freedom. The SPIN [11] model checker was used for this purpose. Protocol design pre-sented in Fig. 3, 4 and 5 was modelled in the PROMELA[5] language. Then the model has been checked for the exis-tence of deadlocks and livelocks. It turned out that all non-progressive cycles [11] 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 [11] (deadlocks) have been found. Besides, the model has been verified to conform to its functional requirements presented in Fig. 6.

3. Integrated formal verification and testing

In this project, automated formal verification is realized by model checking [10]. The NuSMV [9] 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) [10]. Fragmentary example of the NuSMV input specification could be seen in Fig. 7. The model of verified system is an asynchronous statemachine. Its state is described by symbolic variables defined in the module *systems_state* and each transition is represented by corresponding module (e.g. *switch_to_manual*). Module *main* creates all the processes. Model *execution* consists in sequential execution of nondeterministically chosen processes.

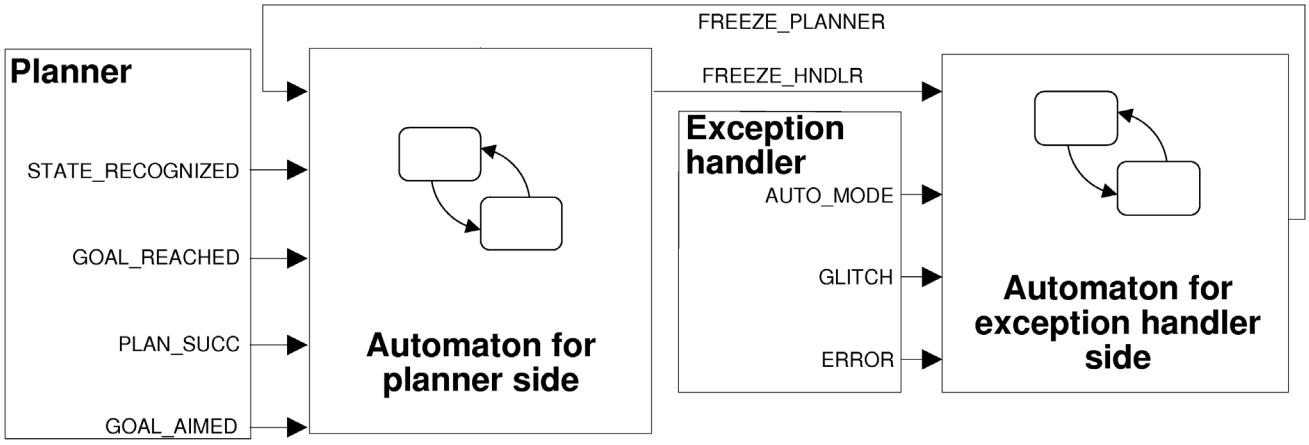

Fig. 3. General scheme of communication between the planner and the exception handler.

4. They should be interpreted according to the operational semantics described in the Stateflow User's Guide [8].
5. A modeling language of the SPIN model checker.

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

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

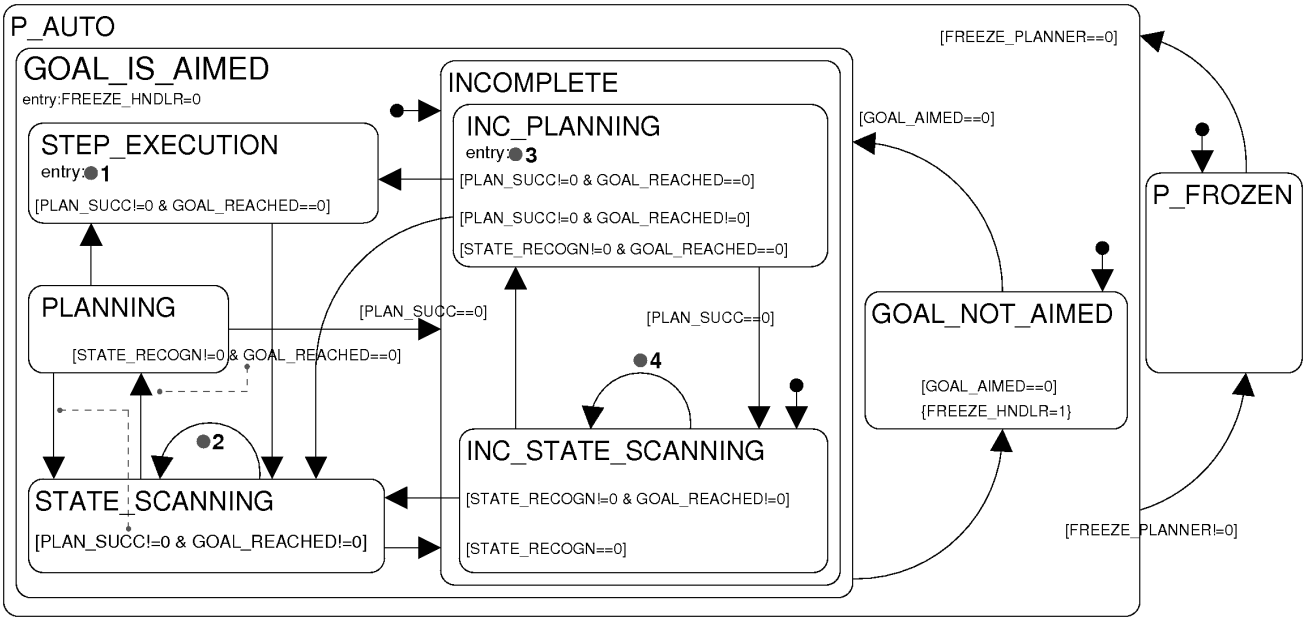

Fig. 4. Design of the communication protocol for the planner.

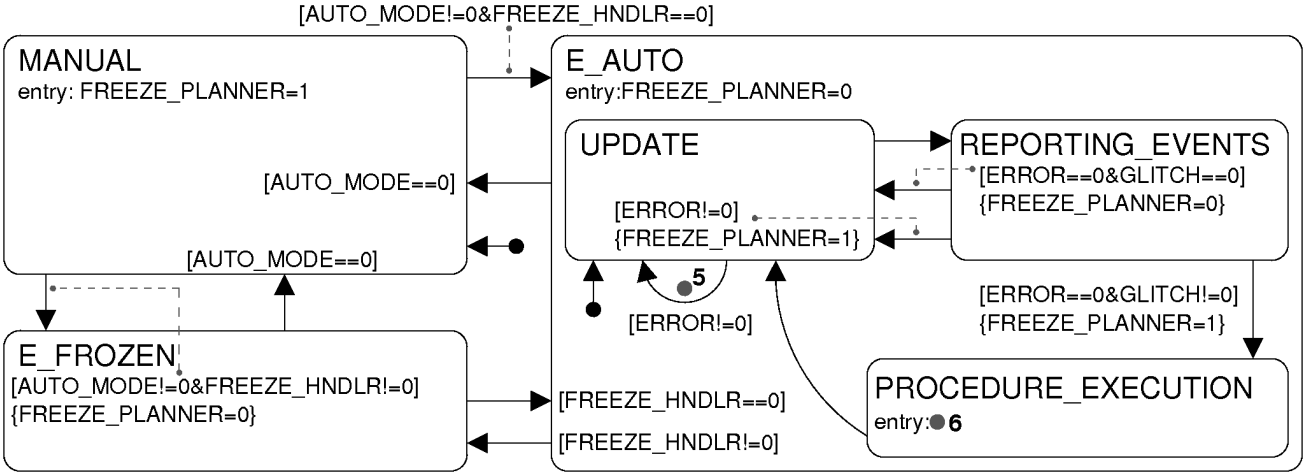

Fig. 5. Design of the communication protocol for the exception handler.

To facilitate the process of automation design, dedicated software has been implemented. The toolbox allows simulating 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.

4. Conclusion

Usefulness of the framework has been evaluated by implementation of supervision software for RF-power station subsystem. This installation is responsible for supplying cavities with energy necessary for particle acceleration. Unfortunately description of this complex installation is beyond of the scope of this paper. It can be

Table 2. Explanation of the state names from the Fig. 4 and 5.

Stimulus name	***Description***
P_AUTO	The planner is permitted to supervise the plant
P_FROZEN	The planner is suspended
GOAL_IS_AIMED	A target operation mode has been specified
GOAL_NOT_AIMED	No target operation mode is specified
STEP_EXECUTION	The planner executes single step of a plan
PLANNING	Planning in progress
STATE_SCANNING	Planner performs state recognition
INCOMPLETE	Planner is incomplete
INC_PLANNING	Planning has failed
INC_STATE_SCANNING	State of the plant is unknown
E_MANUAL	Both engines are suspended
E_FROZEN	The exception handler is suspended
AUTO	The automation engines are permitted to supervise the plant
UPDATE	Exception detection in progress
REPORTING_EVENT	All exceptions are being reported to the operator
PROCEDURE_EXECUTION	Exception handling procedure in progress

1. FREEZE_PLANNER $\rightarrow \Diamond \neg$ FREEZE_PLANNER
2. FREEZE_HNDLR $\rightarrow \Diamond \neg$ FREEZE_HNDLR
3. GLITCH $\rightarrow \Diamond \neg$ P_FROZEN
4. ERROR $\rightarrow \Diamond \neg$ P_FROZEN
5. MANUAL $\rightarrow \Diamond \neg$ P_FROZEN
6. $\neg$ GOAL_AIMED $\rightarrow \Diamond$ GOAL_NOT_AIMED $\wedge$ E_FROZEN

Fig. 6. Properties of the cooperation protocol, which prove its responsiveness and deadlock freedom.

```
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;
          1:                    st.FORCE_MANUAL_MODE;  esac;

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

  -- reachability of MODULATOR_READY
  SPEC EF(state.FORCE_MANUAL_MODE     = FALSE
  &  state.KLY_INTERLOCK_STATUS       = ALL_GREEN
  &  state.MOD_INTERLOCK_STATUS       = ALL_GREEN
  &  state.MODULATOR_STATUS           = ON)
```

Fig. 7. Fragmentary specification for the planner automatically translated to the NuSMV input language.

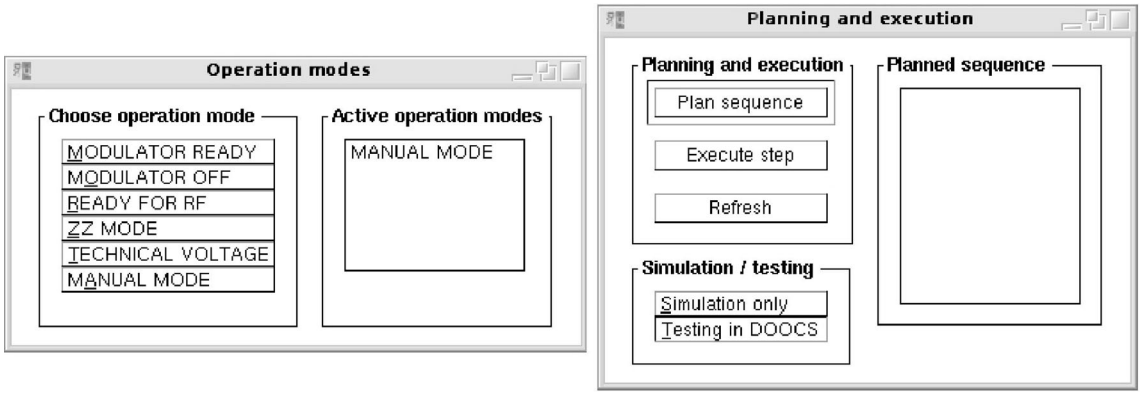

Fig. 8. The interfaces for simulation and step-by-step execution of the automation software.

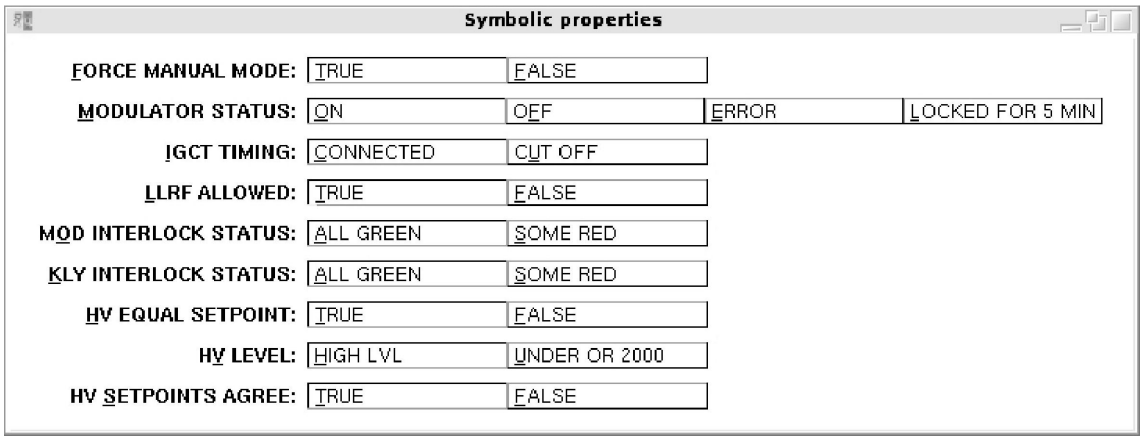

Fig. 9. The graphical user interface for observing and simulating the finite-state model of the planner for the RF power station.

found in [1]. 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[6] active (personal safety, permanent fault).
- RF-leakage[7] 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 fifth RF-power station of the FLASH. It was used to drive the RF-power station to all specified operation modes. It also managed to recover the RF-power station from several field quenches in the accelerating structures (*only RF-inhibit activated*) and modulator faults (*general modulator problem*). These two are the most frequently occurring exceptions, which need to be handled automatically. Response to remaining specified faults was verified by fault injection.

6. Personal interlock indicates potential threat to the personnel servicing the RF-power station. It also indicates presence of the personnel in the vicinity of high power microwave installations.
7. DRF-leakage is a hardware interlock signal reporting leakage of the high power electromagnetic field from the waveguide distribution system, which may cause serious injury of the person subjected to the field.

AUTHORS
Bogusław Kosęda*, Wojciech Cichalewski - Technical University of Lodz, Department of Microelectronics and Computer Science, Al. Politechniki 11, 90-924 Łódź, Poland. E-mail: koseda@dmcs.pl
Tomasz Szmuc - AGH University of Science and Technology, Department of Automatics, Al. Mickiewicza 30, 30-059 Kraków, Poland.
* Corresponding author

References

[1] Aghababyan A., Altarelli M., *et al.*, *XFEL The European X-Ray Free-Electron Laser Technical Design Report*, ISBN 3-935702-17-5, 2006.
[2] Hensler O., Rehlich K., "DOOCS: a Distributed Object Oriented Control System". In: *Proceedings of XV Workshop on Charged Particle Accelerators*, Protvino, 1996.
[3] Ayvazyan V., Rehlich K., Simrock S., Sturm N., "Finite

State Machine Implementation to Automate RF Operation at the TESLA Test Facility". In: *Proceedings of the Particle Accelerator Conference*, Chicago, 2001.

[4] Kosęda B., Cichalewski W., "Design and Implementation of Finite State Machine for RF Power Station". In: *Proceedings of the 12th International Conference Mixed Design of Integrated Circuits and Systems*, Kraków, Poland, 2005.

[5] Kosęda B., Cichalewski W., "Improvements of Expert System for RF-Power Stations". In: *Proceedings of the 13th International Conference Mixed Design of Integrated Circuits and Systems*, Gdynia, Poland, 2006.

[6] Brandt A., Cichalewski W., Koseda B., Simrock S., "Automation of low level RF control operation for the VUV-FEL at DESY and future accelerators". In: *Proceedings of SPIE, Photonics Applications in Industry and Research*, IV 5948, 2005.

[7] Wagner F., *Modeling Software with Finite State Machines: A Practical Approach*, ISBN 0-8493-8086-3, 2006.

[8] MathWorks, Inc., *Stateflow and Stateflow Coder User's Guide*.

[9] Cimatti A., Clarke E., *et al.*, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking". In: *Proceeding of International Conference on Computer-Aided Verification*, Copenhagen, Denmark 2002.

[10] Huth M., Ryan M., *Logic in computer science, Modelling and Reasoning about Systems*, ISBN 0-521-54310-X, 2004.

[11] Holzmann G., *SPIN Model Checker, The: Primer and Reference Manual*, ISBN: 0-321-22862-6, 2004.

[12] Wielemaker J., *SWI-Prolog 5.6 Reference Manual*. Available at: http://gollem.science.uva.nl/SWI-Prolog/Manual/.

[13] *Free On-Line Dictionary of Computing*. Available at: http://foldoc.org/.

[14] Harel D., *Statecharts: A Visual Formalism for Complex Systems Science of Computer Programming*, vol. 8, 1987, pp. 231-274.

[15] Storey N., *Safety Critical Computer Systems*, Addison Wesley, ISBN 0-201-42787-7, 1996.