5th IEEE International Symposium on *High Assurance Systems Engineering* (HASE), Nov. 2000, Albuquerque, New Mexico, USA

Requirements Formalization and Validation for a Telecommunication Equipment Protection Switcher

Michele Cecconi[1]
SIEMENS ICN SPA,
Località Boschetto,
67100 L'Aquila, Italy
michele.cecconi@icn.siemens.it

Enrico Tronci[2]
Area Informatica
Università di L'Aquila
Coppito 67100 L'Aquila, Italy
tronci@univaq.it, http://univaq.it/~tronci

Abstract

Using formal methods*, namely model checking, we can automatically verify a formal model of the requirements against given properties. This allows us to detect errors early in the design process, thus decreasing development cost and time to market. However to modify a well established design process to introduce formal methods is not easy.*

We present a case study exploring the possibility of replacing informal functional specifications with formal ones in the design process of telecommunication Equipment Protection Switchers *(EPSs).*

Our finding is that for EPSs the time effort to write formal specs from informal requirements is comparable with that for writing informal functional specs from informal requirements. This suggests that for EPSs replacing informal functional specs in the design process with formal specs can be done without suffering delays due to the formalization activity.

Key Words Model Checking, Formal Methods, Automatic Validation, Telecommunication Systems, Reactive Systems, Embedded Systems, Finite State Systems, Binary Decision Diagrams.

1 Introduction

Many design errors stem from wrong interpretation of informal requirements. Using a traditional testing approach such errors can only be discovered after an implementation of the system is available so that we can run our tests. However to correct errors at this late stage of development can be costly and time consuming.

One of the goal of *Formal Methods* is to allow detection of errors in the early stages of the design process. In particular by building a formal model of the informal requirements it is possible to detect errors due to wrong requirement interpretation long before we reach the implementation stage. This decreases development costs and time to market. However it may not be easy to smoothly introduce formal methods in a design flow that has been around for years. Even when, as in our case, automatic verification via *Model Checking* is used.

Roughly speaking a typical design flow goes as follows.

1. We are given *informal requirements* describing *what* the system should do. From such informal requirements we get *functional specifications.*
2. *Functional specifications* describe *informally* how informal requirements are to be implemented.
3. Finally from functional specifications an *implementation* is obtained.

The present case study explores the possibility of replacing item 2 in the above sketched design flow with formal specs. That is we explore the possibility of replacing informal (functional) specs with formal specs when informal requirements define telecommunication systems, namely *Equipment Protection Switchers.* This allows us to automatically verify (via model checking) formal specs against given properties. Note that, of course, informal requirements (item 1 above) are not removed from the scene. In our formal setting they are used to write formal specs.

Our main concern is to check that people that are not expert in formal methods can write formal specs within a time comparable with that needed to write informal specs. This will ensure that the design flow will not be delayed by the activity of writing formal specs.

Our case study has been conducted as follows. A student, who only took undergraduate level courses on formal methods, carried out [1] the task of writing and validating (via model checking) formal specs modeling the informal requirements for a *Tributary Equipment Protection Switching* (TEPS) provided by ITALTEL [5].

TEPS is a small system (it only has 2^{18} states). However it contains all the ingredients of larger switchers and well serves our goal of measuring the formalization effort for this kind of systems.

The ITALTEL TEPS project was over (i.e. the system was implemented, tested and delivered) much before we started our case study. This made it possible to compare the efforts related to both approaches (informal vs. formal). Moreover tests were available too. We formalized such tests as formal properties and included them in the properties to be verified via model checking.

We used *First Order Logic on a Boolean domain* (BFOL) to formalize the requirements and to define

[1]This work was done when the author was with ITALTEL SPA

[2]This research has been partially supported by MURST project Tosca.

the properties to be verified. Of course this is just on of the many possible choices. We have an OBDD (*Or dered Binary Decision Diagrams*, [3]) based interprete BSP (*Boolean Symbolic Programming*, [7, 8]) that ca execute BFOL specifications.

The main results of this case study can be summa rized as follows.

- It took one person-month (indeed student-month to write formal specs from the informal require ments for TEPS. This time is comparable wit the time needed to write informal functional spec from the informal requirements.
- ITALTEL provided us with the tests [6] use to test TEPS implementation. Such tests wer aimed at checking the presence of certain tran sitions in the system. Using BFOL we formal ized such tests as *liveness properties* and auto matically, via model checking, verified our forma specs against such properties.

  However by using model checking we can als check the absence of *unwanted* transitions in th system (*safety properties*). To this end we wrot safety properties and checked our formal spec against them.

  The final version of our formal specification o TEPS passed all of our formal properties.
- Our formal specifications can fully replace the in formal functional specifications that are the start ing point for the implementation. This is reason able since, as a matter of fact, our formal specs de fine functional specs using finite state automata This is a familiar approach to our software en gineers since it is essentially the same approac followed by the informal functional specifications It is to be noted, tough, that our formal approac presents automata in a textual form, whereas in formal functional specs use a graphical presenta- tion commented with some (informal) text.

Summing up: our case study suggests that for *telecommunication equipment protection switchers* it is easy and profitable to replace in the design flow informal high level specs with formal high level specs. This results in early error detection (as it happened to us) thus reducing development costs and time to market.

2 Tributary Equipment Protection Switchers

Usually a telecommunication network is built out of units (hardware or software) performing specific functions and cooperating to provide the full system functionality. To reduce the time of loss of service some redundancy is required in the system (redundancy of units). A redundant unit takes the service when the unit that usually provides that service becomes faulty.

A *working unit* is a unit which function is to provide *service* (support telecommunications traffic). A *protection unit* is a unit whose function is to take a service when a working unit becomes failed. A unit is called *active* when it is providing service (regardless of whether it is a working or protection unit); it is called *stand-by* when it is not providing service (because it is a protection unit that has no unit fault to protect or it is a working unit that it is being protected). Changing from active to stand-by and vice versa is called *switching* (of the service).

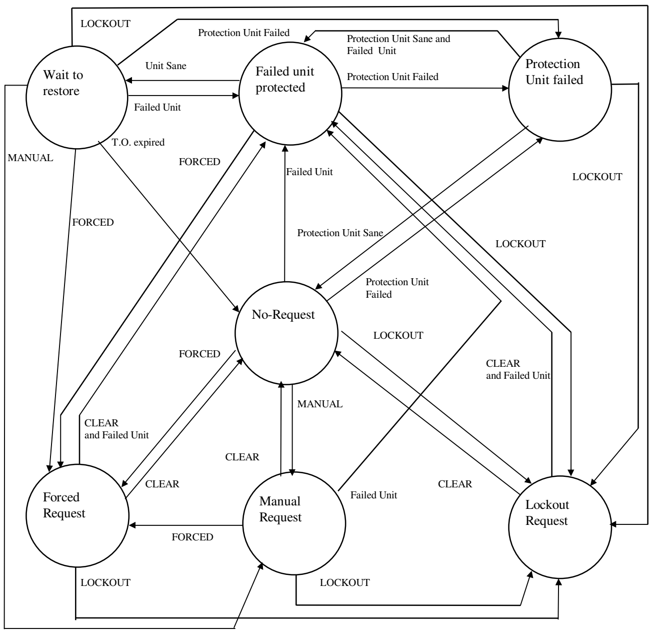

Figure 1: Automaton for group request specification

Generally speaking, it is possible to protect a working unit by means of another identical unit (in this case the protection is called 1:1) or to protect m working units by means of n identical units (in this case the protection is called $m : n$). Note that in both cases the working and protection units are identical. For $m : n$ protection usually $n < m$.

The management of this protection activity is performed automatically, when the system detects a fault on a working unit, or manually, by commands issued by a human operator.

A switching scheme is called *revertive* if when a failed unit is repaired, the protection unit becomes available for protecting other units.

An apparatus implementing the protection activity described above is usually called a *Tributary Equipment Protection Switcher*.

3 Informal Specifications

In this section we sketch some of the informal requirements from [5] that we used as the starting point for our formalization. Our case study refers to a 3:1 equipment protection switcher.

Usually a Protection Switcher can be described as a *Finite State Machine* (FSM) that manages protection requests (from other system components as well as from a human operator). Fig. 1 gives the FSM summarizing our informal requirements.

In what follows commands sent to the switcher are also called *external group requests*. The switcher itself is also called *group*. A *state*, a *condition* as well as the *existing external request* are all components of the *full state* of the system.

3.1 Request Handling

When a new **External Group Request** (EGR) is received it will be compared with the existing request (External Group Request, Condition or State).

If the new EGR has a lower priority than the existing group request, the new EGR will be rejected. If the new EGR has the same or higher priority than the existing EGR, the new EGR will be accepted and stored. (Only one External Request for the protection group will be stored.)

If a new EGR has been accepted and stored or it is issued a Condition or State with a priority higher than the existing (previously stored) EGR, then the previously stored EGR will be removed.

The new request with the highest priority (the newly stored EGR or the newly issued Condition or State) will be executed after the request that is in progress (current request) has been carried out completely, but the *Wait To Restore* (WtR) timer may be interrupted.

If the current group request is *Unit Fail* or *Wait-To-Restore*, a new *Unit Fail* request for another working unit will be accepted, but it will stay pending as long as the current *Unit Fail* condition exists or the WtR timer expires (during this time the protection unit is unavailable because it is providing the service for the former working unit requesting protection).

3.2 Locking

In addition to the active request, each working unit in a 3:1 Equipment Protection Group has either a *Lock Out On* or *Lock Out Off* request active, which provides for inhibiting and allowing protection on a per unit basis.

Multiple units may have *Lock Out On* requests active at the same time.

3.3 Equipment Protection Switch Requests

The following switch requests are provided for the protection group. Each request can be a condition, an external request as well as a state. Requests are listed in descending order of priority. The parameter [unit] issued with the *Forced Switch* or *Manual Switch* represents the working unit to which the command applies. For example, MS(1) switches the service from the working unit 1 toward the protection unit.

3.3.1 Clear

Clear is an External Request.

This External Request cancels the active external request for the protection group.

3.3.2 Lock Out

Lock Out is an External Request as well as a State.

This External Request, with no parameters, is a group request, and prevents further switches of the tributary units until a *Clear* is issued. If the protection unit is active, a switch back to working will be issued.

3.3.3 Forced Switch [unit]

Forced Switch (FS) is an External Request as well as a State.

This External Request switches service to (or maintains service on) protection for the specified unit, which changes from "active" role to "standby" role after a *Forced Switch* has been executed.

Since external requests of the same or higher priority are accepted, a new *Forced Switch* when another unit is already protected by a request of *Forced Switch* will be accepted.

3.3.4 Unit Fail [unit]

Unit Fail is a Condition as well as a State.

The *Unit Fail* condition is issued when a service affecting unit failure has been detected in a unit.

This condition will only result in a switch over to the protection unit if the protection unit is not failing and if the protection unit is not already providing service initiated by a request of the same or higher priority.

The protection unit can also have a *Unit Fail* condition.

The *Unit Fail* on protection has priority over the *Unit Fail* on working.

3.3.5 Manual Switch [unit]

Manual Switch (MS) is an External Request as well as a State.

This external request switches service to (or maintains service on) protection for the specified unit, which changes from "active" role to "standby" role after a *Manual Switch* has been executed.

Since requests of same or higher priority are accepted, a new *Manual Switch* when another unit is already protected by a request of *Manual Switch* will be accepted.

3.3.6 Wait to Restore

Wait to Restore (WtR) is a State.

The *Wait to Restore* state will be issued when the *Unit Fail* condition is no longer valid for a unit that has been protected.

The *Wait to Restore* state will be maintained for a provisioned time.

The service will switch back to the working unit if no unit fail occurs for this working unit during the provisioned time. A request of higher priority will terminate the WtR timer and the higher priority request will be executed, except for the *Clear* external command and for the *Unit Fail* condition issued by a working unit other than the currently protected working unit.

3.3.7 No Request

No Request is a State.

In this state no request is active.

3.3.8 Lock Out [list of all the working unit spec (On/Off)]

Lock Out [list of all the working unit spec (On/Off)] is an external request as well as a State.

This request is not a group request; it is directed toward one or more units, not to the protection group.

The *Lock Out per Unit basis* external request will have the following structure: *Lock Out* [list of all the working unit spec (On/Off)], where the working unit spec parameter will be a list of working units and the switch on/off will be specified for each unit.

This request acts as a toggle for each working unit, allowing and inhibiting protection temporarily for that unit.

When the Request is issued with the switch "off" the unit specified in the working unit spec parameter shows the following behaviour.

If the unit was locked out, the request is accepted and the unit is now allowed protection. *Lock Out Off* is the only way to clear a *Lock Out On* request (i.e. *Clear* has no effect on the *Lock Out On* state).

If the active group request has lower priority than *Unit Fail*, and if one or more of the units which have just been allowed protection have *Unit Fail* conditions, then the lowest numbered unit's *Unit Fail* request becomes the new group request. Otherwise, the group request is not affected.

When the request is issued with the switch "On", the unit specified in the working unit spec parameter shows the following behaviour. If one of the units was in standby state, a switch back to the active state will be issued and a new group request will be determined based on the equipment conditions of the remaining units for whom protection is allowed.

4 Formal Model

In this section we give an outline of our formalization of the requirements in sec. 3.

4.1 Modeling Language

We use BFOL *Boolean First Order Logic* to formalize requirements, as well as properties. Essentially BFOL can be seen as (formal) *scripting* for C based OBDD [3] programming.

Our interpreter from BFOL to OBDD is called BSP (*Boolean Symbolic Programming*) [7].

We define a process (i.e. a *Finite State System*, FSS) with its transition relation. Thus for each process p we have a boolean function (also named p) defining the transition relation of process p. We will denote with **p_px**, **p_nx**, **p_u** the arrays of boolean variables ranging, respectively, on p present states, p next states, p events (actions). We denote with **x** the array of all boolean variables used in our formalization.

4.2 Timer

The process `teps` modeling the requirements for our *Tributary Equipment Protection Switching* is the *synchronous* parallel of two processes: `switcher` (modeling the transition in the switcher) and `timer` (modeling the timer used to issue timeout signals to the switcher).

We start by giving our modeling of the timer. Process `timer` models the timer mentioned in the informal requirements in state `WAITTORESTORE`. We do not have a notion of time associated with our transitions. Thus our `timer` will be a nondeterministic timer that, once started, can nondeterministically stay in a "waiting" state or can "expire". This means that we are asking our system `teps` to work for any time assumptions. This is stronger than needed, but it is easier to check.

The signature for process `timer` is defined in fig. 3. Process `timer` states values and event values are declared using the shorthand `enum` which works similarly to the C `enum` declaration. For example `enum 2 {idle, waiting, expired}` in fig. 3 defines `idle`, `waiting` and `expired` to be, respectively, the following boolean vectors: [0 0], [1 0], [0 1]. That is 0, 1, 2 represented with 2 bits and LSB (Least Significant Bit) on the left. Variables ranging on `timer` present states, next states and events are also defined in fig. 3. We declare them using a C-like syntax. For example `boole timer_u[3]` declares `timer_u` to be an array of 3 fresh (i.e. not previously mentioned) boolean variables (namely: $timer\_u_0$, $timer\_u_1$, $timer\_u_2$). Using an OBDD parlance: `boole timer_u[3]` crates a vector `timer_u` of 3 new OBDD variables.

The transition relation for process `timer` is in fig. 4, whereas in fig. 2 is its graph as a finite state automaton.

4.3 Switcher

In this section we show our model of the switcher.

Process *switcher* has 6 macro states. They are defined using `enum` in fig. 5.

The *switcher* transitions are triggered by 12 *external requests* (events). They are defined using `enum` in fig. 5.

The process `switcher` is defined by giving its transition relation. To this end we need to define (vectors of) boolean variables ranging on `switcher` present states, next states and events (actions).

In fig. 5 are listed the declarations for the variables ranging on process `switcher` events. In fig. 7 are listed the declarations for the variables ranging on process `switcher` present states and next states.

To define process `switcher` it is useful to define a few boolean functions that occur many times in the `switcher` definition. Some of these functions are listed in fig. 6

In fig. 8 we show part of the definition for the process `switcher_clear` defining all transitions of process switcher that are triggered by the external request (event) `clear`. Function $frame_i$ ($i = 1, 2, \ldots$) is automatically added by our interpreter to (the definition of) each transition. It formalizes the *frame condition*, i.e. $frame_i$ says that all state variables for which no update has been defined in transition i will not change their value.

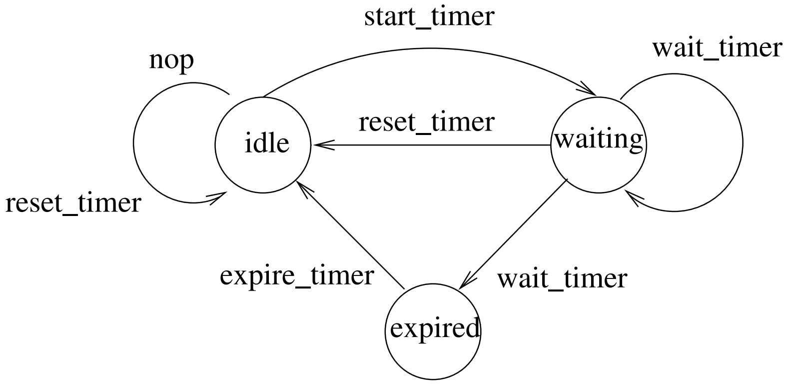

Figure 2: Automaton for process `timer`

`/* timer states */` *enum* 2 {**idle**, **waiting**, **expired**};
`/* timer events */`
enum 3 {**nop**, **start_timer**, **wait_timer**, **reset_timer**, **expire_timer**};
`/* variable ranging on timer events */` *boole* **timer_u**[3];
`/* variables ranging on timer (present [_px] and next [_nx]) states */` *boole* **timer_px**[2], **timer_nx**[2];

Figure 3: States and events for process `timer`

timer(**x**) = (
  ((**timer_u** == **nop**) ∧ (**timer_nx** == **timer_px**))
∨ ((**timer_u** == **reset_timer**) ∧ (**timer_nx** == **timer_px**))
∨ ((**timer_px** == **idle**) ∧ (**timer_u** == **start_timer**)
    ∧ (**timer_nx** == **waiting**))
∨ ((**timer_px** == **waiting**) ∧ (**timer_u** == **reset_timer**)
    ∧ (**timer_nx** == **idle**))
∨ ((**timer_px** == **waiting**) ∧ (**timer_u** == **wait_timer**)
    ∧ (**timer_nx** == **expired**))
∨ ((**timer_u** == **wait_timer**) ∧ (**timer_nx** == **timer_px**))
∨ ((**timer_px** == **expired**) ∧ (**timer_u** == **expire_timer**)
    ∧ (**timer_nx** == **idle**)))

Figure 4: Process `timer`

`/* State values for process switcher */`
enum 3 {**LOCKOUT**, **FORCED**, **UNITFAIL MANUAL**,
WAITTORESTORE, **NOREQUEST**};
`/* External Requests (extreqs) values for process switcher */`
enum 4 {**clear**, **lockout_prot**, **lockout**, **unitfail_prot**, **unitfail**, **forced**, **manual**, **waittimer**, **starttimer**, **expire-timer**, **unit_fixed**, **protection_fixed**};
`/* Variable ranging on external requests sent (from environment) to process switcher (i.e. clear, ... repaired) */`
boole **extreq_name_u**[4];
`/* Defines unit addressed by a unitfail, manual or forced extreq */`
boole **extreq_unit_index_u**[2];
`/*` **extreq_lockout_unit_u**[i] `is 1 if unit` i `(`$i = 1, 2, 3$`) is not in the protection group, 0 otherwise */`
boole **extreq_lockout_unit_u**[4];

Figure 5: Events for process switcher

`/*` $A(\mathbf{x})$ `is 1 iff for all i = 1, 2, 3 it holds: if unit i is not locked out and is failed then unit i is not working */`
$A(\mathbf{x}) = \forall\, i \in \{1, 2, 3\}\ ((\neg lockout\_unit\_px_i \wedge unitfail\_px_i) \rightarrow \neg working\_px_i)$

Figure 6: Auxiliary Functions

```
/* State vars ranging on process switcher state values (i.e.  LOCKOUT ...NOREQUEST).  */
boole switch_px[3], switch_nx[3];
/* 1 if protection unit is in state LOCKOUT (because of a lockout_prot extreq); 0 if protection
unit is not in state LOCKOUT */
boole lockout_protection_px[1], lockout_protection_nx[1];
/* 1 if unitfail_prot extreq has been received (protection unit is not working); 0 otherwise
(protection unit is working) */
boole unitfail_protection_px[1], unitfail_protection_nx[1];
/* Range:  0, 1, 2, 3.  Tells which unit (1, 2, 3) is being replaced by the protection unit.  Value 0
means that no unit is being replaced by the protection unit.  */
boole unit_protected_px[2], unit_protected_nx[2];
/* lockout_unit_px[i] is 0 when unit i (i = 1, 2, 3) can be protected by the protection unit, 1
otherwise */
boole lockout_unit_px[4], lockout_unit_nx[4];
/* unitfail_px[i] is 1 when unit i (i = 1, 2, 3) is not working, 0 otherwise */
boole unitfail_px[4], unitfail_nx[4];
/* working_px[i] is 1 when unit i (i = 1, 2, 3)
is used, 0 otherwise */
boole working_px[4], working_nx[4];
```

Figure 7: States for process switcher

Functions *C1*, *BASE_SWITCH*, *PRCT* are auxiliary functions along the lines of *A* in fig. 6.

$switcher\_clear(\mathbf{x}) = ($
$\quad (\quad (\mathbf{switch\_px} == \mathbf{FORCED}) \wedge \neg A(\mathbf{x}) \wedge BASE\_SWITCH(\mathbf{x})$
$\quad\quad \wedge (\mathbf{extreq\_name\_u} == \mathbf{clear}) \wedge (\mathbf{switch\_nx} == \mathbf{UNITFAIL})$
$\quad\quad \wedge PRTC(\mathbf{x}) \wedge frame_1(\mathbf{x}))$
$\vee (\quad (\mathbf{extreq\_name\_u} == \mathbf{clear}) \wedge (\mathbf{switch\_px} == \mathbf{FORCED})$
$\quad\quad \wedge \neg(\mathbf{lockout\_protection\_px} == \mathbf{unitfail\_protection\_px})$
$\quad\quad \wedge \neg(\mathbf{unit\_protected\_px} == 0) \wedge (\mathbf{switch\_nx} == \mathbf{UNITFAIL})$
$\quad\quad \wedge (\mathbf{unit\_protected\_nx} == 0) \wedge frame_2(\mathbf{x}))$
$\vee (\quad (\mathbf{switch\_px} == \mathbf{MANUAL}) \wedge A(\mathbf{x}) \wedge BASE\_SWITCH(\mathbf{x})$
$\quad\quad \wedge (\mathbf{extreq\_name\_u} == \mathbf{clear}) \wedge (\mathbf{switch\_nx} == \mathbf{NOREQUEST})$
$\quad\quad \wedge (\mathbf{unit\_protected\_nx} == 0) \wedge frame_3(\mathbf{x})) \ldots)$

Figure 8: A glimpse of process `switcher_clear`

$switcher(\mathbf{x}) = (switcher\_clear(\mathbf{x}) \wedge switcher\_lockout\_prot(\mathbf{x}) \wedge$
$\quad switcher\_lockout(\mathbf{x}) \wedge switcher\_unitfail\_prot(\mathbf{x}) \wedge switcher\_unitfail(\mathbf{x})$
$\wedge \quad switcher\_manual(\mathbf{x}) \wedge switcher\_waittimer(\mathbf{x}) \wedge switcher\_starttimer(\mathbf{x})$
$\wedge \quad switcher\_expiretimer(\mathbf{x}) \wedge switcher\_unit\_fixed(\mathbf{x})$
$\wedge \quad switcher\_protection\_fixed(\mathbf{x}) \ )$

Figure 9: Process `switcher`

`/* synchronization between switcher and timer */`
$sync\_switcher\_timer(\mathbf{x}) = ($
$\quad switcher(\mathbf{x}) \wedge timer(\mathbf{x})$
$\wedge \ ((\mathbf{extreq\_name\_u} == \mathbf{waittimer}) \leftrightarrow (\mathbf{timer\_u} == \mathbf{wait\_timer}))$
$\wedge \ (\quad (\quad (\mathbf{switch\_px} == \mathbf{WAITTORESTORE})$
$\quad\quad\quad \wedge \ (\quad (\mathbf{extreq\_name\_u} == \mathbf{lockout\_prot})$
$\quad\quad\quad\quad \vee \ (\mathbf{extreq\_name\_u} == \mathbf{unitfail\_prot})$
$\quad\quad\quad\quad \vee \ (\mathbf{extreq\_name\_u} == \mathbf{unitfail})$
$\quad\quad\quad\quad \vee \ (\mathbf{extreq\_name\_u} == \mathbf{forced})$
$\quad\quad\quad\quad \vee \ (\mathbf{extreq\_name\_u} == \mathbf{manual})))$
$\quad\quad \rightarrow \ (\mathbf{timer\_u} == \mathbf{reset\_timer}))$
$\wedge \ ((\mathbf{extreq\_name\_u} == \mathbf{starttimer}) \leftrightarrow (\mathbf{timer\_u} == \mathbf{start\_timer}))$
$\wedge \ ((\mathbf{extreq\_name\_u} == \mathbf{expiretimer}) \leftrightarrow (\mathbf{timer\_u} == \mathbf{expire\_timer})))$

`/* Tributary Equipment Protection Switcher */`
$teps(\mathbf{x}) = \exists \ \mathbf{timer\_nx} \ \exists \ \mathbf{timer\_px} \ \exists \ \mathbf{timer\_u} \ sync\_switcher\_timer(\mathbf{x})$

Figure 10: Process `teps`

```
/* ******************************************
Automatic switching caused by a ``Unit Fail'' on a working unit.
Initial Conditions:  Normal Operating Conditions.  That is regular service is offered and no alarm is
active.
Test Sequence.
1.  Switch causing conditions leading to declare ``Unit Fail'' on a working unit which is currently
guaranteeing service.
2.  Check switching action and that service stops and then restarts regularly.  However service is now
carried out by the protection unit.
3.  The switcher state must be ``Unit Fail''.
4.  The system is driven back to the starting state by removing the failure on the working unit that
triggered the switching (to ``Unit Fail'').
5.  The switching schema is revertive.  Thus the system enters state ``Wait to Restore'' and service
is carried out by the protection unit until the WtR timer expires (default:  5 min.).
6.  After the WtR timer expires service is interrupted and then resumed regularly.  The system is now
again in normal operating conditions and service is being carried out by the working unit that was
declared failed in step 1.
****************************************** */
```

$$\begin{aligned}
&\mathit{regular1}(\mathbf{x}) = (\ \neg\mathbf{lockout_protection_px} \wedge \neg\mathbf{unitfail_protection_px} \\
&\wedge \neg\mathbf{lockout_unit_px}_1 \wedge \neg\mathbf{unitfail_px}_1 \wedge \neg\mathbf{working_px}_1)
\end{aligned}$$

$$\begin{aligned}
&\mathit{test1_1}(\mathbf{x}) = (\ (\ A(\mathbf{x}) \wedge \mathit{regular1}(\mathbf{x}) \wedge \\
&\qquad ((\mathbf{unit_protected_px} == 0) \leftrightarrow (\mathbf{extreq_u_name} == \mathbf{unitfail}))) \\
&\ \to (\quad (\mathbf{switch_nx} == \mathbf{UNITFAIL}) \wedge (\mathbf{unit_protected_nx} == 1) \\
&\qquad \wedge\ \mathbf{unitfail_nx_1})\ )
\end{aligned}$$

$$\begin{aligned}
&\mathit{test1_2}(\mathbf{x}) = (\mathit{regular1}(\mathbf{x}) \wedge (\mathbf{switch_px} == \mathbf{UNITFAIL}) \\
&\quad \wedge (\mathbf{unit_protected_px} == 1) \\
&\quad \wedge (((\neg\mathbf{lockout_unit_px}_2) \wedge \mathbf{unitfail_px}_2) \to (\neg\mathbf{working_px}_2)) \\
&\quad \wedge (((\neg\mathbf{lockout_unit_px}_3) \wedge \mathbf{unitfail_px}_3) \to (\neg\mathbf{working_px}_3)) \\
&\quad \wedge (\mathbf{extreq_name_u} == \mathbf{unit_fixed})) \\
&\to (\mathbf{switch_nx} == \mathbf{WAITTORESTORE}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{test1_3}(\mathbf{x}) = (\ (\ A(\mathbf{x}) \wedge \mathit{regular1}(\mathbf{x}) \wedge (\mathbf{switch_px} == \mathbf{WAITTORESTORE}) \\
&\qquad \wedge (\mathbf{unit_protected_px} == 1) \wedge (\mathbf{extreq_name_u} == \mathbf{expiretimer})) \\
&\ \to (\quad (\mathbf{switch_nx} == \mathbf{NOREQUEST}) \wedge (\mathbf{unit_protected_nx} == 0)))
\end{aligned}$$

$$\mathit{test1}(\mathbf{x}) = (\mathit{teps}(\mathbf{x}) \to (\mathit{test1\_1}(\mathbf{x}) \wedge \mathit{test1\_2}(\mathbf{x}) \wedge \mathit{test1\_3}(\mathbf{x})))$$

Figure 11: Liveness property formalizing test 1p

For each external request we have one process defined analogously to `switcher_clear`. The transition relation for process `switcher` (fig. 9) is the logical *and* ($\wedge$) of the transition relations for such processes.

4.4 Tributary Equipment Protection Switcher

The definition of the transition relation for process `teps` modeling the *Tributary Equipment Protection Switcher* defined in the informal requirements is obtained from the *synchronous* parallel of processes `switcher` and `timer`. This is shown in fig. 10. Note the synchronization with process `timer` to model the requirements in state `WAITTORESTORE`.

5 Verification

To validate our formal model `teps` of the informal requirements we carried out automatic verification (via model checking) of two kind of properties: liveness properties (stating that certain transitions must be in `teps`) and safety properties (stating that certain transition should not be in `teps`).

5.1 Liveness Properties

Liveness properties aim at checking the presence in our formal model `teps` of transitions that are present in the requirements. We obtain our liveness properties by formalizing the tests used by ITALTEL to test their implementation of TEPS [6].

For example the informal statement for test 1p is in fig. 11. Test 1p can be formalized with boolean function `test1` in fig. 11 (function `A` has been defined in fig. 6).

Our BSP interpreter checks if formula `test1` is identically 1 (true). That is if property `test1` is true for all assignments of the boolean variables from which `test1` depends. If this is the case then `teps` satisfies property `test1` (i.e. `teps` passes test 1p).

It took just a few days to formalize all tests, and a few seconds to verify (via model checking) all of our system properties.

Our first requirement formalization of `teps` passed all liveness properties but three of them. Such failing properties pointed out misinterpretation of a few points in the informal requirements. Using the feedback from our attempt to verify such failing properties we corrected our `teps` modeling. The resulting version of `teps` passed all of the liveness properties we ran (via model checking).

```
/* ************************************************
Check priority of ``Unit Fail'' on protection unit w.r.t.  ``Unit Fail'' on group unit.

Conditions.
1.  The protection unit is not working, i.e.  the switcher is in state UNITFAIL.
2.  The protection unit cannot offer protection to any failed group unit.
3.  Under the above conditions service cannot be switched from a group unit towards the protection
unit without using a ``forced'' external request.
************************************************ */
```

$$\begin{array}{l} \textit{testneg2}(\mathbf{x}) = \\ (\quad (\quad \textit{teps}(\mathbf{x}) \wedge (\textit{switch\_px} == \mathbf{UNITFAIL}) \\ \qquad \wedge (\neg \mathbf{lockout\_protection\_px}) \wedge \mathbf{unitfail\_protection\_px} \\ \qquad \wedge (\mathbf{unit\_protected\_px} == 0) \wedge (\neg(\mathbf{extreq\_name\_u} == \mathbf{forced})) \\ \qquad \wedge (\neg(\mathbf{extreq\_name\_u} == \mathbf{lockout\_prot}))) \\ \ \rightarrow ((\mathbf{unit\_protected\_nx} == 0) \wedge (\mathbf{switch\_nx} == \mathbf{UNITFAIL}))) \end{array}$$

Figure 12: Safety property formalizing test 2n

5.2 Safety Properties

Safety properties aim at checking that behaviours that are ruled out (explicitely or implicitely) by the requirements are not in our formal model `teps`.

For example the informal statement for property 2n is in fig. 12. Property 2n can be formalized as in fig. 12.

It took just a few hours to write such safety properties and a few seconds to model check all of them.

Some safety properties (among which is property 2n) were not passed by our `teps` model, even when all liveness properties were passed.

This required minor modifications in our `teps` model as well as in the formulation of our safety properties. For example many of our safety properties initially failed just because we did not ruled out the external request *forced*.

Our present formulation of `teps` passed all of our liveness as well as safety properties.

5.3 Experimental Results

Process `teps` has only 2^{18} states, so it is quite small. It takes less than 3 seconds to build the OBDD representing `teps` and to verify all properties we were interested in. So, as we said, modeling effort was our main concern here rather than state explosion.

6 Conclusions

We presented a case study about requirement formalization for telecommunication *Equipment Protection Switchers* (EPSs).

Our main concern was to compare, for EPSs, the effort needed to write formal specs from informal requirements with that needed to write informal functional specs from informal requirements.

The time spent writing formal specs (1 person-month) turns out to be comparable with that spent writing informal functional specs. Thus, at least for the kind of systems we studied, replacing informal specs with formal specs in the design cycle appears to be easily accomplishable without major time delays or personnel retraining. Moreover using a formal approach it is also possible to validate (the formalization of the) requirements against liveness and safety properties.

References

[1] C. Antonelli, *Uso di Metodi Formali per la Validazione Automatica di Specifiche*, Master's Thesis in Computer Science, University of L'Aquila, L'Aquila, Italy, July 1998

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang, *Symbolic model checking: 10^{20} states and beyond*, Information and Computation 98, (1992)

[3] R. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. on C., Vol. C-35, N.8, Aug. 1986

[4] E. A. Emerson, *Temporal and Modal Logic*, Handbook of Theoretical Computer Science, Elsevier 1990

[5] ITALTEL, *2.0Mbit/s Tributary Equipment Protection Switching: System Requirements*, 1994

[6] ITALTEL, *2.0Mbit/s Tributary Equipment Protection Switching: System Verification*, 1994

[7] E. Tronci, *Hardware Verification, Boolean Logic Programming, Boolean Functional Programming*, Proc. 10th IEEE Conf. on "Logic In Computer Science" 1995, San Diego, CA, USA

[8] Tool available at url: `http://univaq.it/~tronci/bsp.html`