Logical Consistency Validation Tools for Distributed Systems

Original Scientific Paper

Drago Žagar
J. J. Strossmayer University of Osijek,
Faculty of Electrical Engineering
Kneza Trpimira bb, Osijek, Croatia
drago.zagar@etfos.hr

Nino Vrandečić
Hrvatska elektroprivreda, Osijek
nino.vrandecic@hep.hr

Antun Stoić
J. J. Strossmayer University of Osijek,
Faculty of Mechanical Engineering in Slavonski Brod
Croatia
astoic@sfsb.hr

Abstract *– As a result of using Information Technology (IT) in different technological processes it is necessary to develop new application specific communication protocols. The number of application specific protocols is growing rapidly in different areas: medicine, communication, industry, power systems, computer networks, etc. Protocol errors discovered in the implementation phase are usually a consequence of inconsistent protocol design, which implies the necessity of methodology for error detection in an early design phase. This paper describes formal methods for distributed systems, especially SPIN/Promela tool for formal verification of logical consistency in distributed systems. A protocol used in power systems IEC60870-5-101 has been verified as an example of formal verification of a distributed system. Formal specification, simulation and verification of logical consistency have been successfully done by using SPIN/Promela software.*

Keywords *– distributed systems, finite state machine, protocol, validation*

1. INTRODUCTION

The processes within distributed systems communicate by protocols defined according to the processes needs and specific properties as well as the characteristics of transmission paths between the processes. A result of the recent IT technology impact on different technologies is development of new application specific communication protocols. The number of application specific protocols is increasing very fast in different and diverse technology fields. A specific protocol is a very complex product that should be attuned to user demands and error-free. Protocol errors could be found in every phase, but the most "expensive" for correction are final phases of protocol design. As the errors discovered in the coding and implementation phase are very often the consequence of inconsistent design, it is desirable to find a method for error detection in an early phase of protocol development. To ensure early error detection the problems have to be formally defined on an abstract level.

An increased complexity of new (especially embedded) systems and new system development trends show an increased level of multidisciplinarity a consequence of which is that system development methods used in one discipline could be successfully used in many others. The most important trends in the development of multidisciplinary systems are increasing complexity (requirements, hardware and software components, interfaces, etc.), increasing integration and test effort and increasing time to market pressure. All of these elements influence the lower level of error tolerance. Some authors use formal models for formal verification of multidisciplinary embedded systems (e.g., the Promela model enabled formal verification of the SoCoMo system). A result is a more flexible and cost effective developed process [1]. Another application of formal methods is in design and analysis phases of industrial production systems (and especially in so-called Flexible Manufacturing Systems, FMSs) as a complementary tool to current practice of the field. Some projects showed that the Flexible Manufacturing approach yielded a deeper understanding of the finer points of system specification, an improved system reliability (through verification of desired properties), and other typical engineering qualities [2][3][8].

This paper describes formal methods for distributed systems and the SPIN/Promela tool for formal specification and verification of logical consistency. A possible usage of formal methods in other disciplines is verified by specification and verification of a protocol used in power systems IEC60870-5-101. Formal specification, simulation and verification of logical consistency have been successfully done by using SPIN/Promela software.

2. FORMAL SPECIFICATION OF DISTRIBUTED SYSTEM

FSM - Finite State Machine is one of the first formal models used for formal specification of sequence machine behavior, the machines whose states do not depend only on input states but also on input states history. The finite state machine is not a heuristic model that could be differentially interpreted and has very good theoretical background. The finite state machine is a very intuitive method and therefore designer friendly.

On the lowest abstraction level, protocols could be described as machine states. In the process of protocol design, it is possible to specify acceptable and non acceptable protocol states easily as well as the transitions between states according to input events. A solution for specification of a complex distributed system is problem apportion into well-defined abstract and less complex machine states. This simplified machine states communicate through interfaces.

The use of machine states will be most effective if communication is realized as emulation of real distributed system. This means that communication could be synchronous or asynchronous. Asynchronous communication connotes machine states connection by the FIFO (First-In First-Out) channel (Figure 1.)

The signals of machine states are represented by abstract objects called messages. The input signal is taken from the input tail while the output signal is sent to the output tail. Synchronization is achieved by setting the conditions for input and output signals.

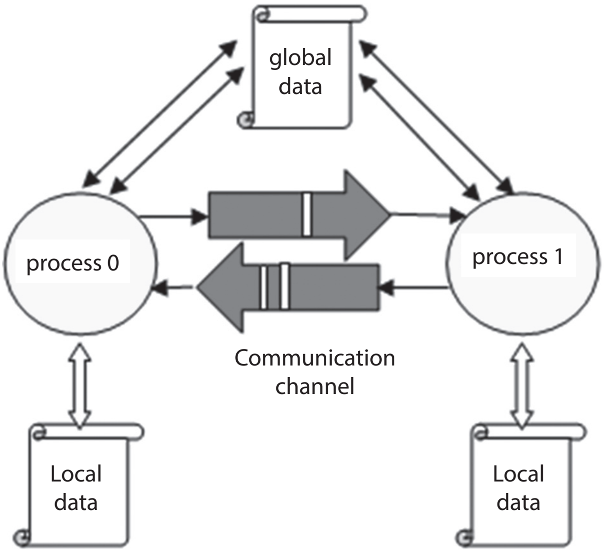

Fig. 1. Communication between machine states defined by processes

For every state, one or more transition rules that are executable could exist. If only one transition rule is executable, it means deterministic transition to a new state. If no transition rule is executable, the machine will go to the final state.

Hence the distributed system consists of many machines communicating with each other, and the whole system could acquire diverse state/behavior combinations. Some of the combinations could be undesirable. The desirable system behavior is defined by basic system design. The system without demands cannot be correct and we can say that such system behavior is unpredictable. The relationship between desirable system states could be visually presented by sets intersection of all possible states with the set of undesirable (irregular) states. The system is valid if this intersection is an empty set, or mathematically [6]:

$L(S)$ – set of possible states from S

$L(p)$ – is a set of valid (desirable) states

If fulfilled:

$L(S) \cap L(\neg (p)) = \emptyset$,

or if an intersection of possible and undesirable states if empty, then we can say that the system is error-free, or more exactly, the system works according to the defined, required behavior. It is important to emphasize that absolute assertion about system validity is not possible. It could be only stated if the system fulfills the specific criteria.

Some of the basic criteria for proving compatibility with demands (validity criteria) are as follows:

- the system should not have deadlocks – invalid end state,
- the process should not starvate any other process,
- no explicit assertion inside the process should be transgressed,

- all processes should terminate properly (end state),
- the system should be effectively progressive.

Therefore, the distributed system is described by two formal entities, i.e. by:

- system specification,
- specification of demands on system behavior.

Both entities together create a verification model, and a process of system compatibility determination is called the verification model of the distributed system. To create both entities we need a high-level abstract language for system design. The program written in such language is called the validation model. There is a small difference between model validations and model implementation [6].

3. FORMAL TOOLS FOR DISTRIBUTED SYSTEM VERIFICATION

There are several tools for formal verification of logical consistency in the distributed system. One of the best that is also free of charge is SPIN. The model in SPIN is written by Promela language (PROcesses Meta Language). Promela does not prevent improper and inconsistent design but enables verification of design by using the "model checker". Promela defines three types of basic objects: asynchronous processes, global and local objects and messages channels. When Promela does not define the global system clock, synchronization is realized by global variables and channel messages [6][7].

A core of Promela is instruction executability: every instruction is foregone by a condition, followed by a consequence. If the condition is not fulfilled, the instruction is not executable (it is blocked) and it does not have the consequence (e.g., it means that the instruction is executable if we have q messages m (q?m) in the channel). If the channel is empty or if it contains another message, the process must wait.

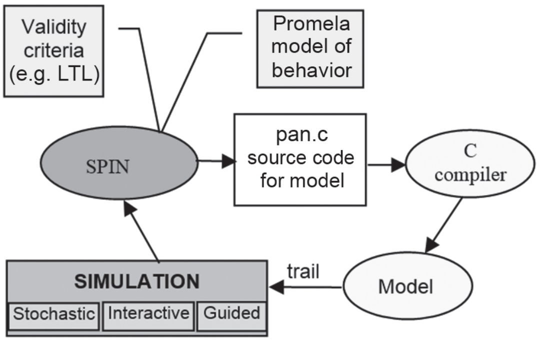
Validity criteria (e.g. LTL)
Promela model of behavior
SPIN
pan.c source code for model
C compiler
Model
trail
SIMULATION
Stochastic
Interactive
Guided

Fig. 2. SPIN configuration

Every process has its local state consisting of local variables values and a program state counter, while the whole model is described by the global state vector consisting of the values of global variables, messages channel content and a list of active processes. The processes could start and stop at any time, but they leave the state vector only by LIFO order. The process is deleted from the state vector by two steps: termination and process death. The "parent" processes must terminate before the "children" processes. The initialization process (init) terminates the last.

Specification of the system and demands on the system SPIN are carried out in one file, but it is possible to create criteria based on linear temporal logic formulae (LTL) and start process verification without changes in the system model (Figure 2).

In addition to the basic criteria, SPIN also uses the definition of non-desirable behavior called the never claim. The never claim can be built in the system model or generated from the LTL formula by the LTL property manager. The LTL formula defines desirable and non-desirable system behaviors, and SPIN generates the never claim and translates desirable behaviors in non-desirable and vice versa, depending on which sort of behavior is selected as validity demand.

Besides verification, SPIN could be used for simulation of the distributed system. This option could help us get the visual perception on system validity. Simulation could be run in three modes:

- stochastic simulation (if transition is non-deterministic, SPIN stochastically chooses one of them),
- interactive simulation (the user chooses the path),
- guided simulation (in the case of error by verification, SPIN creates trail error).

The SPIN verification model is based on the analyses of machine states reachable from the initial state.

According the number of states that we have to analyze, SPIN uses one of three possible methods:

- Exhausting search analyzes compatibility of validation criteria in all system states.
- Controlled partial search – Supertrace. Supertrace will be run in case an available memory is less than needed for coverage of all machine states
- Stochastical search is used for huge systems by which neither Supertrace gives satisfactory results.

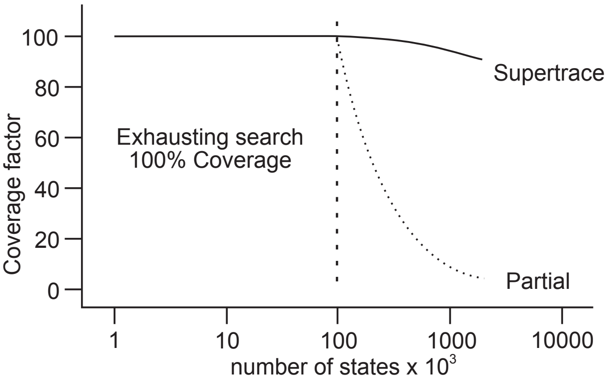

Fig. 3. Coverage factor and search modes

The quality of search is expressed by the coverage factor of machine states, defined as a ratio of the analyzed and the total number of states (Figure 3). By exhausting search the coverage factor is 100%. Switching to uncontrolled search the coverage factor decreases, as well as the quality of analyses. A measure of search quality describes verification method ability to find the error, and it is defined by the ratio of founded and total errors [6][7][8].

4. DESIGN OF EXPERIMENT

As a case study, we will implement the formal protocol specification and verification method on protocol IEC 60879-5-101 used in power systems. In this section, the case study protocol will be described with basic communication characteristics [4][5].

4.1. COMMUNICATION REGIMES

IEC 60870-5-101 is a very robust protocol for information exchange in the power system, specifically for communication of remote terminal units, i.e., RTU. IEC 60870-5-101 is built on three layers of the OSI model, physical, data link and application. It describes communication between the control center and distant stations.

The communication could be arranged in two modes, i.e., balanced and imbalanced. By balanced mode, both sides could initiate communication as peer entities. Therefore, for every single station we must have a separate communication channel. By imbalanced mode, communication is based on the master-slave principle. Only the control center (master station – primary station) can initiate communication. The slave station (secondary station) can only respond to requests received from the master station.

Simultaneously, only communication between primary and one secondary station is possible. Communication is arranged by polling.

4.2. BASIC COMMUNICATION SERVICES

A data link enables three basic services:

- SEND / NO REPLAY – the primary station does not expect any answer from the secondary station. It is used for setup in round robin manner, broadcasting, and time synchronization.
- SEND / CONFIRM – the primary station expects answer from the secondary station. If the answer fails, the primary station can repeat the message. CONFIRM message could be only one character (xE5) or a small message NACK or ACK.
- REQUEST / RESPOND – this service could be used by procedures for application data transmission from the secondary station (User data or Requested data not available) and also by procedures for link status establishment (Status of link). This service is not used by balanced communication.

4.3. LOCAL INITIALIZATION OF THE PRIMARY STATION IN IMBALANCED MODE

Initialization of the primary station is carried by bootstrapping of the control center (on/off). The process of initialization starts by reestablishment of communication between communication sides. The primary station initializes a connection by request "request status of link". The secondary station answers positively by "status of link" or negatively by "access denied". The next step is to reset the link "reset remote link". After primary station initialization there follows the "general interrogation" sequence.

4.4. LOCAL INITIALIZATION OF THE SECONDARY STATION

Local initialization of the secondary station starts after bootstrapping (e.g., for maintenance purposes). If the primary station is active, it will detect the secondary station answer failure. The primary station retransmits the same request for a preset number of attempts. If after that the secondary station does not answer, the primary station will try to re-establish a link by "request status link". If the secondary station is initialized before the primary station exploited all requests for link status, the secondary station answers by link status followed by link reset as described in Section 4.3. To establish a connection between the application in the primary and the secondary station, the secondary station sends a message "end of initialization".

4.5. DISTANT INITIALIZATION OF THE SECONDARY STATION

Distance initialization of the secondary station is performed by a command "reset process command". The secondary station resets all its application processes, and abandons all previous messages for sending. After that, on receiving a request for data of class 2 the secondary station answers by reset acknowledgement. The next step is activation of link reset, described in the former section. By terminating the reset process, the secondary station answers the request for data of class 1 (2) by the end of the initialization message. Distant initialization of the secondary station is used only for maintenance purposes.

4.6. DATA ACQUIRED BY IMBALANCED COMMUNICATION MODE

IEC 60870-5-101 defines two data classes. Data class 1 is defined for spontaneous events (e.g., a change of single and double indication), while data class 2 specifies periodically acquired data (e.g., measurements). The normal communication procedure proceeds as follows:

- The primary station sends requests for data class 2 (request user data class 2). If the secondary station does not have any event from class 2 (periodical event), it will send negative acknowledgment (NACK), and the primary station will poll a succeeding secondary station.

- • If the secondary station has a spontaneous event from class 1, it will reply negatively to the request of class 2 and in control byte it will change the ACD (access denied) bit into 1. After that, the primary station issues request for data class 1, and the secondary station replies by data of class 1, depending on ASDU organization and data amount the data could be sent in one frame of variable length. If not all data can be sent in one frame class 1, then the secondary station will repeat with frames of class 1 until all events are sent. The last frame of the secondary station will notify by ACD=0. Afterwards, the primary station starts polling of class 2.
- If only spontaneous data are transferred (no measurements), then it is more efficient to use class 2.
- If the secondary station continuously generates on the application level, it could also be a consequence of error state and the other secondary stations will be blocked. Therefore, the secondary station limits the maximal number of consecutive requests of class 1.

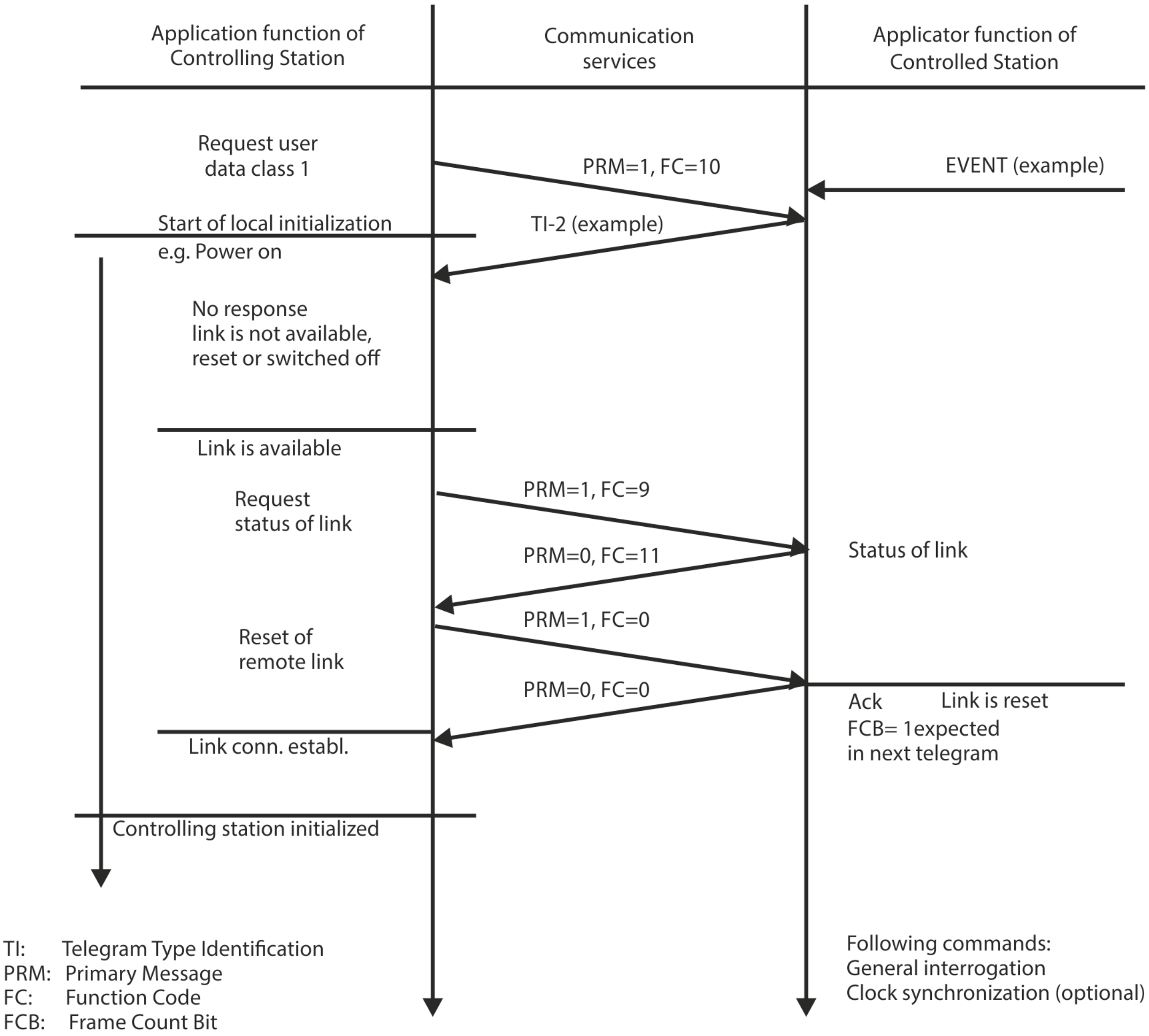

Fig. 4. Local initialization of primary station

4.7. GENERAL INTERROGATION

The function general interrogation is used for full update of process data after the primary station detects data loss (unsuccessful timeout after many retransmissions). The general interrogation function undergoes procedures of local/distant initialization of the secondary station and local initialization of the primary station. Interrogation data are sent to the primary station in one or more ASDUs according the amount of process data. A procedure ends with end interrogation notification (TI100/COT=10). The interrogation procedure could be interrupted if spontaneous events happen.

5. RESULTS OF EXPERIMENTS

In this section, formal tools will be applied for specification and verification of the primary and the secondary station in the IEC 60870-5-101 protocol. In a modeling process, we will start from the level of implementation which is partially described in the following section [4][5][7][9].

5.1. GLOBAL VARIABLES

A core of the model consists of two processes, i.e., a process specifying primary station behavior - PRMstation, and a process specifying secondary station behavior - SECstation. These two processes communicate by two channels:

- channel *to_rcvr* for data transmission from primary to secondary station, declared as: *chan to_rcvr = [0] of { mtype, bit,bit };*
- channel *to_sndr* for data transmission from secondary to primary station, declared as: *chan to_sndr = [0] of { mtype, bit,bit };*

A communication between processes is synchronous and defined by 0 channel capacity

Channel messages are of type mtype carrying two binary parameters.

Mtype is defined as:

mtype = { ReStatLink, StaOfLink,ResRemLink, ACK, ReqClas2, ReqClas1, NACK, DataClas, error,ReqClas2GI, GIData, TI100 , TI100GIEnd};

with following semantics:

ReStatLink –the primary station requests link status from the secondary station.

StaOfLink – the secondary station replies to the primary station by link state information.

ResRemLink – the primary station requests link reset from the secondary station

ACK – the secondary station replies positively to message type SEND/CONFIRM

ReqClas2 – the primary station requests class 2 data.

ReqClas1 - the primary station requests class 1 data.

NACK - the secondary station replies negatively to message type SEND/CONFIRM.

DataClas – represents user data, i.e., ASDU message code in direction monitoring. This variable is attached the value on the application level and symbolizes a process value.

error - is a symbol variable which represents an error by communication between the primary and the secondary station.

ReqClas2GI – this variable represents a message by which the primary station requests interrogation data from the secondary station. This is an abstraction because the implementation does not distinguish separated requests for class 2 in the interrogation process. Upon a unique request for data of class 2 the secondary station replies by ASDUs with COT (the cause of transmission) field set to dec20 (active interrogation state) or dec3 (interrogation not active). In our model, we will assume that the secondary station does not send the information about the data back, but only single request ReqClas2GI defines what the primary station requires. According to that, the received data will be treated as the data from the interrogation table.

GIData - a variable representing part of the interrogation table, which could be sent in one ASDU. (limitation of the frame length is 255 bytes).

TI100 - a variable representing a command issued by the primary station to the secondary station in order to start the interrogation process

TI100GIEnd – by terminating the interrogation process, the secondary station informs the primary station.

Every message is assigned two parameters type bit. The first parameter is the ACD (*access demand*) bit. It is set to 1 in the process SECstation in case the process detects class 1 data. In all other cases, this bit is set to 0. The second bit is FCB (*frame count bit*). In the implementation model, the primary station changes this bit by every new transmitted frame. The Promela model is slightly different. By the Promela model, it is defined that the primary station sends this bit to the secondary, which returns the same bit. In the implementation model, the secondary station does not return FCB but successively analyses all values of this bit. If two successive bits are the same, the secondary station assumes the error in command direction (from the primary to the secondary station), rejects the frame and sends NACK. For simplicity reasons, the Promela model will not take into account the error that occurred in command direction, but it will only model the error in monitoring direction. Therefore, the FCB in the primary station is used to repeat the same message as a last not replied message.

Secondary station application processes are represented by three channels. *Channels eventClass1* and *eventClass2* use 1 and 2 as event sources of classes. Channel *G1* is used as a source of interrogation data. In order for event channels to be ready (filled) before the processes of primary and secondary stations start, channels *eventClass1, eventClass2,* and *G1* should be initialized by the *atomic* sequence inside the initialization process *init:*

```
init {
atomic{          /* initialises the channels of class 1
and 2 with data for simulation */
eventClass1!DataClas (1);
eventClass1!DataClas (2);
eventClass1!DataClas (3);
eventClass1!DataClas (4);
eventClass2!DataClas (10);
eventClass2!DataClas (20);
eventClass2!DataClas (30);
GI!GIData(1);
GI!GIData(2);
}
}
```

Channel *chan pogreska = [1] of {mtype}* for the information exchange between the command process and the primary station process.

General variable *count* generates stochastic events by simulation.

```
mtype = { ReStatLink, StaOfLink,ResRemLink, ACK, ReqClas2, ReqClas1, NACK, DataClas, DataClas1, DataClas2 ,error,ReqClas2GI, GIData, TI100 , TI100GIEnd};
chan to_sndr = [1] of { mtype, bit,bit };
chan to_rcvr = [1] of { mtype, bit,bit };
chan eventClass1 = [10] of {mtype};
chan eventClass2 = [10] of {mtype};
chan GI = [10] of {mtype};
chan error = [1] of {mtype};
bit ACD, LinkStatus;
int count;
```

5.2. PROCESSES

The model defines six processes:

- *PRMstation* specifies the primary station (control center),
- *SECstation* specifies the secondary station (remote terminal unit),
- *EventMaker* generates events of class 1 and 2,
- *Watchdog* detects timeouts in the model,
- *RandGen* generates a random number,
- *Init* is the basic initialization process.

PRMstation:

The primary station start by starting a procedure of local initialization of primary station LinkStatus==0. After that, a sequence starts in which the primary station polls the secondary station by requesting the link status (to_rcvr!ReStatLink). If the secondary station replies positively (to_sndr?StaOfLink) the primary station, a procedure of link reset starts (to_rcvr!ResRemLink). If the secondary station replies positively, it connotes that the link is established, LinkStatus==1. After finishing the link reset procedure, the primary station requests general interrogation (to_rcvr!TI100) from the secondary station. The secondary station replies positively. If the primary station receives an acknowledgement ready for general interrogation (to_sndr?ACK), the primary station successively issues requests for GI data and receives data (to_sndr?GIData). After that, the interrogation table (in our model – channel G1) is empty, the secondary station sends the message end of interrogation, and the primary station receives (to_sndr?TI100GIEnd). During the execution of this loop, it is possible that the following condition is also fulfilled:

```
:: to_sndr?DataClas(0,FCBRtu);
:: pogreska?error -> LinkStatus = 0; goto progress
```

The first condition will be fulfilled if during the interrogation procedure an event happens. The result is that the interrogation procedure will be temporarily interrupted while all events are sent. The second condition will be fulfilled in case the control process (watchdog) detects timeout. That results in stopping the interrogation request and repeating the procedures for link status and link reset.

If during interrogation all data are correctly transmitted and LinkStatus==1, the primary station issues requests of class 2 (to_rcvr!ReqClas2) until message *to_sndr?NACK*(*1,FCBRtu*) is received, where the first parameter specifies the *ACD* bit. The next step specifies *ACD==1* and the primary station issues request class 1. The requests are repeated until the secondary station sends a message with *ACD==0*. After that, class 2 requests will be repeated.

If an error occurs while the primary station requests data class 2 (ACD==0) or class 1 (ACD==1) (secondary to primary station direction), it is possible to make a nondeterministic choice:

```
1)    :: pogreska?error ->  goto FindClass2
2)    :: pogreska?error -> LinkStatus=0; goto progress
```

The first case assumes that after a preset number of timeouts the primary station repeats a request of class 2 or class 1, until receiving a reply. After a certain number of unsuccessful requests, the primary station will request the status of link. By simulation the choice can be chosen randomly or by the user.

The secondary station process behaves as a server process, i.e., it sends back the requested data on request. The process has one main loop (loop 1) in which the process runs continuously. After receiving a request for general interrogation and a positive reply, the process enters loop 2 in which it stays as long as the data in channel *G1* exist. After channel *G1* is empty, the control is returned to loop 1. Loop 2 could also be interrupted in case of error. The error is defined by a nondeterministic choice, e.g., if during the interrogation process there exist data of class 1, Promela could randomly read and send the data from channel eventClass1 to the primary station or do nothing but leaving the loop. In this way, the message loss from the secondary to the primary station is simulated, and the secondary station is set to the state ready to receive new messages declared in loop 1. Errors by requests of class 1 and 2 are simulated similarly.

RandGen:

A generator of random numbers randomly increases or decreases *counter* count for step 1. This process is used only when we use the process of randomly generated events.

EventMaker:

This process on the base of global variable *count*, This process that is based on global variable count, which changes depending on process RandGen, adds events in channels of class 1 and 2. The following form never terminates but waits indefinitely while variable *count* coincides with the predefined value. If we want to limit the number of generated events, the process can terminate leaving the loop by command *break*.

Watchdog:

This process generates the message about the error on channel *pogreska* in case any active process in the model detects *timeout*.

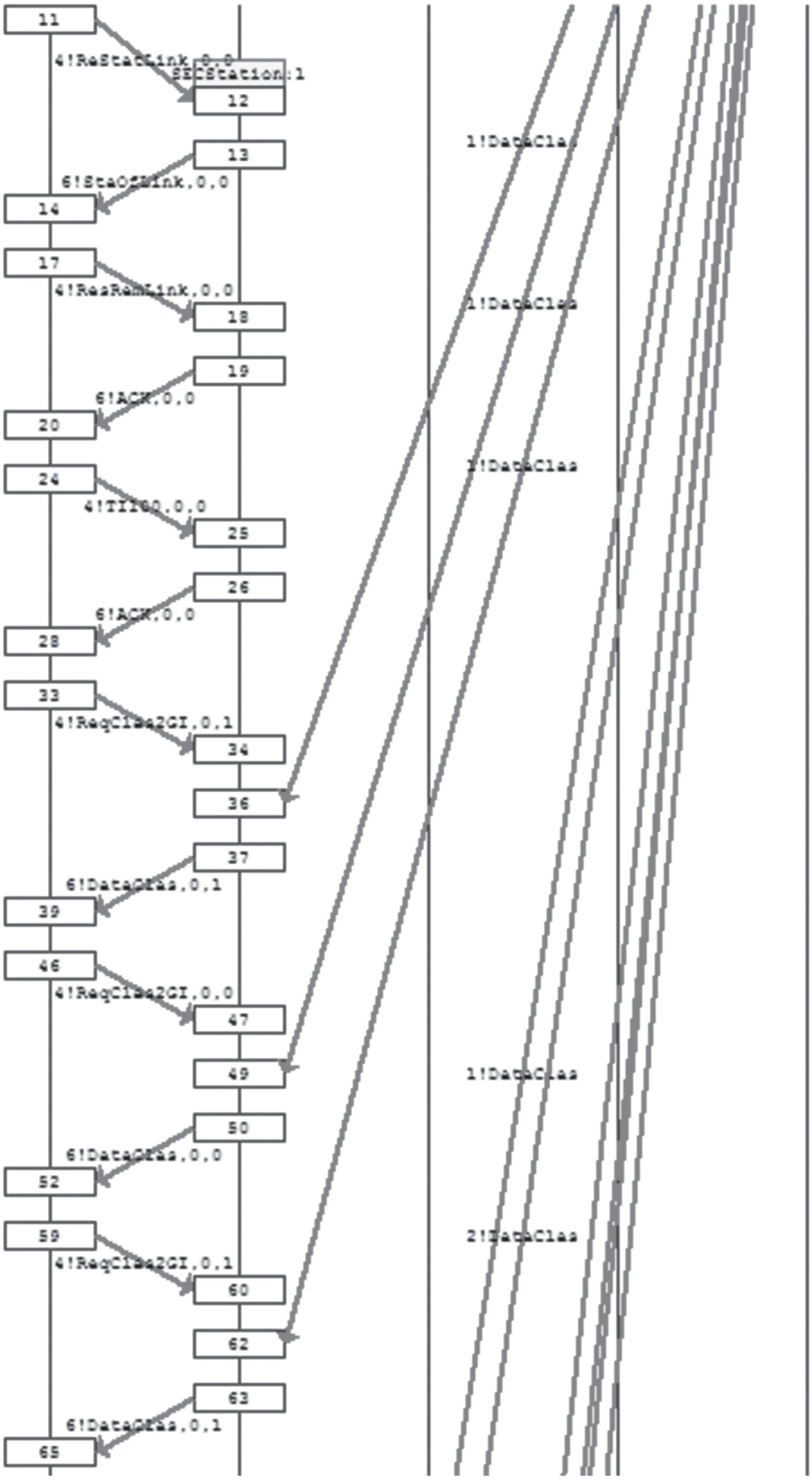

Fig. 5. Message sequence chart by local initialization of primary station

5.3. MODEL SIMULATION

Simulation of the specified model could be performed by one of three methods described in Section 3. Simulation enables detection of coarse design errors, especially by models with size enough that a designer could have a deep insight into the problem. During simulation, it is possible to track all local and global variables as well as the channel's messages in the window showing the actual values. The simulation archive is stored in three ways:

- *Simulation output* shows a chronological simulation trace.
- *Message sequence chart* graphically describes the simulation sequence written in simulation output (Figure 5). Every process is assigned one vertical thread. The transition between the states is shown by a directed arrow, while every transition execution is shown by a sequence number as the one in simulation output.
- *Time sequence diagram* shows a simulation run similarly to thread execution [9][10][11].

5.4. MODEL VERIFICATION

Model verification is done by respecting the following criteria:

- The secondary station process could be without termination, progressive only if the primary station is progressive. Therefore, we can expect that the secondary station waits for demands in two points: endSEC and endSEC2, i.e., at the beginning of one of the two loops inside the process. These points in the Promela model could be treated as points of regular process termination.
- On the contrary, the primary station should be constantly progressive, which is specified by the main loop *progress* and the progress of loops for finding class 1 and 2, *progressFindClass1* and *progressFindClass2*. If the secondary station does not reply to messages, proper process termination could be treated a point in the primary station process by which the primary station repeats the requests for link status. As the model is done in a way that this is not allowed, the primary station should be verified only respecting the progress.
- Verification is done separately, on demand of the progress (non-progress cycle) and on demand of proper termination (invalid end state).

Verification results:
active verification on non-progress cycles

```
Full state space search for:
   never claim              +
   assertion violations     + (if within scope of claim)
   non-progress cycles      + (fairness disabled)
   invalid end states       - (disabled by never claim)
State-vector 100 byte, depth reached 297, errors: 0
   2973 states, stored (3590 visited)
   1127 states, matched
   4717 transitions (= visited+matched)
   352 atomic steps
   hash conflicts: 6 (resolved)
Stats on memory usage (in Megabytes):
   0.321 equivalent memory usage for states
   (stored*(State-vector + overhead))
   0.491  actual memory usage for states (unsuccessful
   compression: 152.79%)
State-vector as stored = 157 byte + 8 byte overhead
   2.097  memory used for hash table (-w19)
   0.320  memory used for DFS stack (-m10000)
   0.158  other (proc and chan stacks)
   0.081  memory lost to fragmentation
   2.827  total actual memory usage
```

Verification results:
active verification on invalid end states

```
Full state space search for:
never claim              - (not selected)
assertion violations     - (disabled by -A flag)
cycle checks             - (disabled by -DSAFETY)
invalid end states       +
State-vector 96 byte, depth reached 152, errors: 0
2201 states, stored
174 states, matched
2375 transitions (= stored+matched)
328 atomic steps
hash conflicts: 0 (resolved)
Stats on memory usage (in Megabytes):
0.229 equivalent memory usage for states
(stored*(State-vector + overhead))
0.388  actual memory usage for states (unsuccessful
compression: 169.58%)
State-vector as stored = 168 byte + 8 byte overhead
2.097  memory used for hash table (-w19)
0.320  memory used for DFS stack (-m10000)
0.158  other (proc and chan stacks)
0.081  memory lost to fragmentation
2.724  total actual memory usage
```

Verification results prove that every *label* labeled by prefix progress is infinite and all processes could "terminate" in states labeled as *label* and *prefix* end (verification result does not have any error).

By the first verification, the SPIN analyzed 3590 states, 1127 of which were reanalyzed, and the total number of transitions is 4717. The longest executable sequence was 297 steps long.

By the second verification, the SPIN analyzed 2201 states, 147 of which were reanalyzed, and the total number of transitions is 2375. The longest executable sequence was 152 steps long.

By both verifications, reachability analyses of system states were carried out, which was also the basic request. Unreachable states of the adjacent process are expectable and could be explained.

The verification model could be executed based on requirements defined by linear temporal logic (LTL). For verification of this model, we used the following requirements:

- at least once, it should happen that the primary station issuing the link status request does not receive a reply. This presumes the error in monitoring direction or the secondary station is switched off or out of order. The request is written as (<> !j), where #define j(to_sndr?[StaOfLink]).
- at least once, LinkStatus should be equal to zero. This means that it could not happen that the program is progressive while the process of link establishment is bypassed and LinkStatus is by definition equal to zero. The request is written as (<> g) , where #define g(LinkStatus ==0).
- As the secondary station is basically a nondeterministic model, a reply to the link status request issued by the primary station is expected at least once. The request is written as (<> j), where #define j (to_sndr?[StaOfLink]).
- After the link is established, it is expected that the primary station will at least once receive an NACK message, interrogation data and end of interrogation message. The request is written as (p-> (<> h)) && (p -> (<> dataGI)) && (p-> (<> mTI100)), where #define p (LinkStatus==1), #define dataGI (to_sndr?[GIData]), #define mTI100 (to_sndr?[TI100GIEnd]).
- at least once, an error should happen while LinkStatus==0. This presumes error possibility in monitoring direction while the link is established. The request is written as (<> (pog U g)), where #define pog (pogreska?[error]).

Taking into consideration all requirements, the LTL formula reads:

(<> !j) && (<> g) && (<> j) && (p-> (<> h)) && (p -> (<> dataGI)) && (p-> (<> mTI100)) && (<> (pog U g))

During preprocessing of the Promela model by macro functions, logical variables will be assigned the following global variables:

```
#define p       (LinkStatus==1)
#define h       (to_sndr?[NACK])
#define g       (LinkStatus ==0)
#define j       (to_sndr?[StaOfLink])
#define pog (pogreska?[error])
#define dataGI (to_sndr?[GIData])
#define mTI100 (to_sndr?[TI100GIEnd])
```

On the basis of the defined LTL formula, the SPIN creates a *never claim* process. *Never claim* could be built in the basic Promela model and verified together with other validity requirements or verification could be performed only for the *never claim* request. If verification of never claim is performed independently (by LTL property mAnager), it is necessary to remove *never claim* from the basic Promela model. From verification results we can conclude that there is no violation of requirements defined by the LTL formula.

6. CONCLUSION

An increased complexity of new mostly distributed systems and new development trends show an increased level of multidisciplinarity. A consequence is that system development methods used in one discipline could be used in many others. The trends in development of multidisciplinary systems show increased complexity, integration level as well as time to market pressure. All of these elements influence the lower level of error tolerance.

A wide application of distributed systems, especially communication protocols, requires some methods of automatic verification. The system errors are usually a consequence of an inconsistent protocol design. The errors discovered in an early phase of system development result in less time wasting and lower financial cost implying a necessity for methodology of error detection in an early design phase.

This paper describes formal methods for distributed systems and SPIN/Promela tool for formal specification and verification of logical consistency. The possible usage of formal methods in other disciplines is verified by specification and verification of a protocol used in power systems IEC60870-5-101. Formal specification, simulation and verification of logical consistency have been successfully done by SPIN/Promela software. The case study results did not detect any termination process error or deadlock state error. The processes are effectively progressive and verification proves that every *label* labeled by prefix progress is infinite and all processes could “terminate” in states labeled as *label* and *prefix* end (the verification result does not have any error). From verification results, we can conclude that there is no violation of requirements defined by the LTL formula that confirms compatibility with specific requests of temporal logic. The processes did not have any unexplainable and unreachable state.

7. REFERENCES

[1] N.C.W.M. Braspenning, Reducing the lead time of developing multi-disciplinary embedded systems by model-based integration, Technische Universiteit Eindhoven, 2006.

[2] N. Trčka, Verifying Chi Models of Industrial Systems with SPIN, Proc. of International Conference on Formal Engineering Methods, No8, LNCS 4 260 Macao, China, 2006.

[3] A. Matta, et.al., FM for FMS: Lessons Learned While Applying Formal Methods to the Study of Flexible Manufacturing Systems, Proc. of 4th Internacional Conference on Theoretical Aspects of Computing , LNCS 4711, Macau, China, September 26-28, 2007, pp. 366-380.

[4] Norwegian IEC 870-5-101 Approved version Revision no 2. Oslo, 1998.

[5] SINAUT LSA Control Center Interface Module IEC 60870-5-101 Telecontrol Profile – Manual. Siemens, 1999.

[6] G. J. Holzmann, Design and Validation of Computer Protocols, Murray Hill, New Jersey, USA, 1991.

[7] G. J. Holzmann, D. Peled, An Improvement in Formal Verification, Proc. of FORTE Conference, Bern, Switzerland, October 4-7 1994, pp. 1-12.

[8] N. Guelfi, A. Mammar, A Formal Approach for the Verification of E-Business Processes with PROMELA, Software Engineering Competence Center, University of Luxembourg, Technical report, 2004.

[9] G.J. Holzmann, The Model Checker Spin, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

[10] http://www.stateworks.com, accessed: May 2013.

[11] http://spinroot.com/, accessed: April 2013.