УДК 004.414.38

Developing formal temporal requirements to distributed program systems

Shoshmina I. V. (Peter the Great Saint-Petersburg Polytechnic University)

Developing temporal requirements to distributed program systems an engineer should determine and systemize event sequences caused by system processes interleaving. A number of such sequences grow exponentially that makes the requirement development procedure nontrivial. This is why engineers prefer not to construct or construct elementary formal requirements. As result powerful formal verification methods become unavailable or some important properties of distributed systems leaved unexpressed. While it is well-known, that development of formal requirement even without verification improves an quality of a distributed system structure and functions.

In this paper we suggest a method for formal temporal systems development which is easy-to-use. The method is based on scalable patterns of linear temporal logic formulas.

Using this method we developed formal temporal requirements to a practical program control system (a vehicle power supply control system). Verifying the requirements with the model checking method we found 3 critical errors that were missed by developers of the vehicle power supply control system during design and testing.

Keywords: software requirement specification, requirement patterns, model checking, linear temporal logic

1. Introduction

Developing temporal requirements to distributed asynchronous program system is complicated in practice. Because an engineer should systemize an exponential number of system behaviour sequences resulting as process interleaving.

The wide-spread approach to solve this problem is to use formulas patterns for requirements: an engineer tries to find a requirement close to a pattern. Dwyer et al. in [1] developed the *specification pattern system* (SPS). They analyzed 500 temporal requirements to systems from different application fields and suggested patterns for the most typical ones. The main SPS drawback it is too strict: patterns aren't scalable to different events number.

De-facto SPS has become the standard [2], [8], [9], [10]. Later it was modified by different way. In [2], [3] patterns were extended by real time and probabilistic requirements. In [4], [5] there were suggested nested patterns for interval logic, in [6] — nested patterns for linear

temporal logic. In [2], [7] authors described patterns in limited English.

In spite of these modifications the strict structure of SPS formulas has left unchanged. In this research we develop patterns that, on one hand, could be scalable and, on another hand, we give an easy way how to use these patterns to develop significant temporal requirements. As a base for our patterns we use the temporal relation “leads-to” [11] where after an environment stimulus somewhen in the future there should follow a system reaction. Our patterns are formalised in the linear temporal logic (LTL).

Using our patterns we developed and verified formal temporal requirements to a power vessel supply control system (PVSS). The PVSS were developed and provided to us by a Russian ship control systems manufacturer. The original PVSS code was written on C++ and contained 20000 code strings (not counting external libraries). One of the most complicated PVSS characteristics was an asynchronous work of its modules. Verification allows us to find 3 critical errors that developers did not find neither during design nor during bench and program testing.

2. Patterns of events sequences

Temporal requirements of program systems are often some event sequences. The most suitable and concise temporal logic for describing event sequences is the linear temporal logic (LTL). LTL–formulas consist of atomic propositions $p \in AP$, Boolean operations and temporal operations: Until – $\mathcal{U}$ and the Next time – $\mathcal{X}$ (NextTime). This is grammar for a LTL–formula φ:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathcal{X}\varphi \mid \varphi\,\mathcal{U}\,\varphi \tag{1}$$

To short fomulas we will use some extra operations, Boolean ($\Rightarrow$, $\wedge$, etc.) and temporal ones: Future – $\mathcal{F}$, Globally – $\mathcal{G}$, Release – $\mathcal{R}$, Weak Until – $\mathcal{W}$, where $\mathcal{F}\,\varphi = \top U \varphi$; $\mathcal{G}\,\varphi = \neg\mathcal{F}\,\neg\varphi$, $\varphi\,\mathcal{R}\,\psi = \neg(\neg\varphi\,\mathcal{U}\,\neg\psi)$; $\varphi\,\mathcal{W}\;\psi = \mathcal{G}\,\varphi \vee \varphi\,\mathcal{U}\,\psi$, and the true constant: $\top = p \vee \neg p$. We use a common formal semantics of LTL formulas defined on infinite sequences (i. e. [12]).

A lot of formulas decribing different temporal requirements could be constructed with LTL. We consider one that has very practical application, when a system environment gives a stimulus by an event s and then the system guarantees an event–reaction p somewhen in the future (p *is after* s). We call the relation as the “unconditional response”. It’s a LTL formalization $Resp(s,p)$:

$$Resp(s,p) = s \Rightarrow \mathcal{F}\,p. \tag{2}$$

The requirement *"If someone from a floor calls an elevator then in the future the elevator will stop at that floor"* is an example of a requirement with the unconditional response. The response relation in the form (2) is well–known as "leads–to" [11].

Now we require that a system should remember receiving a stimulus s until emitting a reaction p by setting a condition t. So we get a "conditional response" relation denoted it as $Resp(s, p, t)$:

$$Resp(s, p, t) = s \Rightarrow t\,\mathcal{U}\,p. \tag{3}$$

Similarly we define a conditional precedence relation (*before an event–reaction p should be an event–stimulus s which sets a condition t*), denoted as $Prec(s, p, t)$:

$$Prec(s, p, t) = \neg p \Rightarrow (t\,\mathcal{U}\,p \Rightarrow \neg p\,\mathcal{U}\,s). \tag{4}$$

The requirement *"If a fire fighting system switched on then before that a duty officer gave its a corresponding command"* is an example of a requirement with the precedence relation.

Formulas (3) and (4) describe local temporal relations between a stimulus and a reaction in sense that a temporal relation is satisfied in a current state of a system behaviour. To develop requirements we should consider temporal relations (3) or (4) in all states of a system behaviour. Let's consider 4 typical systems work phases: *start*, *global*, *regular*, *final*. In a *global* phase a temporal relation should be satisfied in all system states. Other phases define a scope where a temporal relation is satisfied. In a *final* phase a temporal relation should be true after the final phase started; in a *start* phase — before the phase finished; in a *regular* phase a temporal relation should be satisfied during the phase. Defining phases bounds by events we get following LTL formulas for temporal requirements:

$$\begin{aligned} global(s, p, t; \varphi) &= \mathcal{G}\,\varphi(s, p, t), \\ fin(s, p, t; q; \varphi) &= \mathcal{F}\,\mathcal{G}\,q \Rightarrow \mathcal{F}\,global(s, p, t; \varphi), \\ start(s, p, t; r; \varphi) &= \neg r \Rightarrow \varphi(s, p, t)\,\mathcal{W}\,r, \\ reg(s, p, t; q, r; \varphi) &= \mathcal{G}\,(q \Rightarrow start(s, p, t; r; \varphi)), \end{aligned} \tag{5}$$

when φ is substituted by a formula $Resp$ or $Prec$ from (3)–(4), the formula $global$ defines a requirement in a global phase, formulas fin, $start$ and reg — for final, start and regular phases respectively. The variable q defines an event of starting a final phase in fin, r — an event ending a start phase in $start$, and variables q and r – events starting and ending a regular phase respectively.

The suggested temporal relations (3)–(4) are so that they are easily scalable to stimuli and reactions consisting from event sequences (not from one event): $\vec{s} = \{s_1, s_2, \ldots, s_m\}$, $\vec{p} = \{p_1, p_2, \ldots, p_n\}$ with sequences of conditions $\vec{v} = \{v_1, v_2, \ldots, v_{m-1}\}$, $\vec{t} = \{t_1, t_2, \ldots, t_n\}$, restricting stimuli and reactions respectively:

$$\begin{aligned}
\mu(\vec{s}, \vec{v}) &= s_1 \wedge v_2 \mathcal{U} (\ldots s_{m-1} \wedge (v_m \mathcal{U}\, s_m) \ldots), \\
\chi(\vec{p}, \vec{t}) &= t_1 \mathcal{U} (p_1 \wedge \ldots t_{n-1} \mathcal{U} (p_{n-1} \wedge (t_n \mathcal{U}\, p_n)) \ldots), \\
Resp(\vec{s}, \vec{p}, \vec{v}, \vec{t}) &= \mu(\vec{s}, \vec{v}) \Rightarrow v_2 \mathcal{U} (s_2 \wedge \ldots (v_m \mathcal{U} (s_m \wedge \chi(\vec{p}, \vec{t}))) \ldots), \\
Prec(\vec{s}, \vec{p}, \vec{v}, \vec{t}) &= \neg p_1 \Rightarrow (\chi(\vec{p}, \vec{t}) \Rightarrow \neg p_1 \mathcal{U}\, \mu(\vec{s}, \vec{v}))
\end{aligned} \qquad (6)$$

In this case requirements expressed in LTL formulas (5) aren't changed except substituting φ by formulas $Resp$ or $Prec$ from (6) and adding $\vec{v}$.

If requirements depended on an infinite behaviour of environment we describe them by the following LTL formula:

$$\psi \Rightarrow \xi, \qquad (7)$$

when ξ is a formula from (5), ψ — a formula defining an infinite environment behavior. Common fairness requirements is a particular case of the formula (7).

Comparing patterns of [1] with ours ones by temporal relations structure we could resume that 83 formulas from 217 LTL-formulas of [1] have a response relation $Resp$, 13 formulas — a precedence relation $Prec$, while other 121 don't contain relations between a stimulus and a reaction. So our LTL–pattern coincide with existing practical requirements and even allow scaling them to represent wider temporal dependencies.

3. Developing requirements to the power vessel supply control system

The considered power vessel supply system (PVSS) consists of two power supply stations (PS) while a power supply station contains a diesel, a generator, a generator cutout switch (GCS). The power vessel supply control system coordinate the work of these PSs. Its structure is shown at the fig. 1 inside the bold frame. We will use the abbreviation PVSS for the power vessel supply system and for its control system. All PVSS controllers have independent asynchronous behaviors coordinated by passing messages. Environment modules/devices the PVSS works with are drawn outside the frame. The PVSS monitors and controls these devices

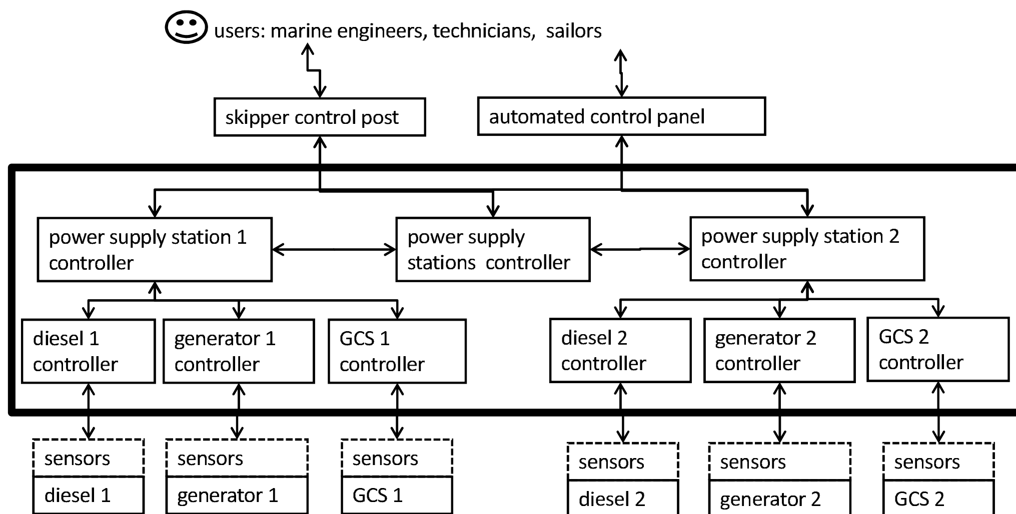

Рис. 1. The PVSS Structure

by reading sensors values and setting signals. A diesel has the utmost number of sensors (12 pieces).

The PVSS provides electricity power to all vessel consumers. For that it dynamically switches on/off power stations depending on loading. The PVSS activity could be quite complicated. For example, to start a power station the PVSS starts a diesel at first. When the diesel rotation becomes stable, PVSS starts a generator, after that it starts a generator cutout switch. And only after that consumers get the electricity power. Moreover, procedures of switching power stations on/off depend on PVSS modes and could be different.

"PSSV requirement specification" provided us by PVSS developers was written quite poor and did not contain enough information about PVSS to develop formal temporal requirements. So we used mainly "Bench testing program and technique". The test from this manual is cited at the fig. 2. To resolve ambiguity and uncertainty we used "Operating guide', the PVSS code too, and sometimes consulted with experts developed the PSSV.

At first we identified input/output events from tests in natural language (like at the fig. 2). We will use some of them in requirements below.

To combine events into temporal requirements sequences we will use the patterns (5)–(6). If someone would like to avoid the direct usage of formulas he/she could use the modified problem frame approach and translate requirements to formulas from graphical problem frames [13]. In general modified problem frames allow to construct temporal requirements unlike the original one developed by M. Jackson [14].

The test at the fig. 2 describes the PVSS transition to a remote automated mode. Analysing

D.1.2.1 Before start:

- DG1 and DG2 stopped (banners "DG1 is ready to start" and "DG2 is ready to start" lighten in the ACP window "Power Supply Station");
- the SCP switch "PS mode" is in the position AUT;

D.1.2.2 Testing transition to the PS remote automated control mode.

D.1.2.2.1 Switch on the test bench "Testing PS control algorithms" buttons "DG1: ready to start", "DG2: ready to start", "DG1: remote control on", "DG2: remote control on", "SCP control".

D.1.2.2.2 That should have the following effects:

- on the ACP display the message "PS control mode — remote" received and indicators "Control mode — remote", "DG1 is ready to start", "DG2 is ready to start" changed to yellow;
- on SCP lamps "DG1: SCP control", "DG2: SCP control", "DG1: ready to start", "DG2: ready to start" lighted up.

D.1.2.3 Testing results are accepted when all effects described above happened.

Рис. 2. Testing transition to a PVSS remote automated mode. Abbreviations: DG – diesel generator, ACP – automated control panel, SCP – skipper control post, AUT – automated.

other tests of "Bench testing program and technique" we found out that the PVSS could transfer to the remote automated mode independently of diesels state. This is why the test at the fig. 2 splits to few temporal requirements, in particular: "Transition to a remote automated mode", "Diesel 1 activation in the remote automated mode", "Absence of a diesel 1 misactivation in the remote automated mode" and symmetrical for the diesel 2.

Transition to a remote automated mode. *Always when the PVSS is not in the remote automated mode and it would be in this mode in the future then before that an operator gives commands "DG1: remote control on", "DG2: remote control on" on the ACP and changes the switch "PS mode" in the position AUT on the SCP.*

The requirement is written in LTL so:

$$\mathcal{G}\,(\neg dist \wedge \mathcal{F}\, dist \Rightarrow \neg dist\,\mathcal{U}\, autosig), \tag{8}$$

where *autosig* — the signal to set the remote automated mode (commands "DG1: remote control on", "DG2: remote control on" and the switch "PS mode" in the position AUT), *dist* — the signal that the remote automated mode is set.

The other temporal requirement describes an absence of an unwanted diesel 1 activation.

Absence of a diesel 1 misactivation in the the remote automated mode. *Always in the remote automated mode the lamp "DG1: ready to start" wouldn't light up until the diesel 1 is ready to start.*

The temporal relation in this requirement corresponds to the precedence pattern (6), so as result we get:

$$\mathcal{G}\,(dist \;\Rightarrow\; ((\neg readylamp \wedge \neg hand\,\mathcal{U}\,readylamp) \Rightarrow \neg readylamp\,\mathcal{U}\,ready)\,\mathcal{W}\,hand), \tag{9}$$

where $ready$ — the signal from sensors that the diesel 1 is ready to start (simulated at the fig. 2 as the ACP banner “DG1 is ready to start”), $readylamp$ — the lamp “DG1: ready to start” lights, $hand = \neg dist$ — manual or local modes is set, $dist$ — as in (8).

The diesel 1 in the test at the fig. 2 activated (becomes ready to start) if the remote automated mode is set enough long. This is modelled in LTL as “somewhen forever”.

The diesel 1 activation in the the remote automated mode. *If somewhen forever the remote automated mode is set up then somewhen forever the lamp “DG1: ready to start” would light up to the sensors signal that the diesel 1 is ready to start.*

Formally:

$$\mathcal{F}\,\mathcal{G}\,dist \Rightarrow \mathcal{F}\,\mathcal{G}\,(ready \Rightarrow \neg hand\,\mathcal{U}\,readylamp), \tag{10}$$

where $dist, hand, readylamp, ready$ — the same are in (9).

At the tab. 1 we compare our formal temporal requirements development to PVSS and “Bench testing program and technique”. As result, we described more events explicitly than it was in an events table of “Bench testing program and technique”. We found out requirements that unnecessary repeated in different tests. We defined requirements that were formulated implicitly, for example, the requirement (9) is implicit in the test at the fig. 2. So we resume that developing formal temporal requirements with the patterns (5)–(6) gives a better structure of requirement specification than informal procedures. But some requirements described by quite complicated LTL formulas containing 10-15 events.

4. Verifying the power vessel supply control system

We claim that our patterns allow to describe important requirements to distributed programs. To approve that we verified the PVSS with respect to developed formal temporal requirements using SPIN [15]. At first we constructed a PVSS model in Promela, the input language of SPIN. A PVSS module algorithm was modeled as an independent asynchronous process. Processes coordinated their work transferring messages by asynchronous channels.

Table 1

Comparing formal requirement development method and bench testing program on the PVSS

	Formal requirement development method	Testing program
Number of explicitly enumerated events	71	20
Number of requirements or tests	36 requirements	23 tests
Average size of a requirement or a test	10-15 subformulas (precedence relation), 30-36 subformulas (response relation)	2 pages (A4)
Development time	2 weeks	unknown

Because the PVSS model was large we reduced it manually. To check our temporal requirements we used 4 reduced models of the PVSS model. Correctness of reduced models is proved by correspondence of counterexamples traces in Promela with traces in the original C++ code.

Let's consider one of the critical error found out in the PVSS verification. Because this error obviously shows problems that developers of program systems meet with, and such errors are quite difficult to analyze and understand without verification.

Starting a reserve diesel–generator while another one crashed *If the power station 2 hardware failures infinitely often, and the power station 1 hardware works properly infinitely long, and always in case of failure of the power station 1 hardware the protection would be reset and the remote automated mode with the power station 2 priority is set, then somewhen in the future for every reserve response the diesel–generator 1 would start.*

Formally the requirement is so:

$$\bigwedge_i \mathcal{G}\,\mathcal{F}\,\tilde{b}_i \wedge \bigwedge \mathcal{F}\,\mathcal{G}\,\neg b_i \wedge \mathcal{F}\,\mathcal{G}\,\neg reset \wedge \bigwedge_j \mathcal{G}\,(b_j \Rightarrow \mathcal{F}\,reset) \wedge \mathcal{F}\,\mathcal{G}\,prior21 \quad \Rightarrow \\ \mathcal{F}\,\mathcal{G}\,(reserv \Rightarrow prior21\,\mathcal{U}\,lampon), \tag{11}$$

when $\tilde{b}_i$ – sensors data of the diesel–generator 2, b_i – sensors data of the diesel–generator 1, $reset$ – reset a protection, $prior21$ – the remote automated mode with the power station 2 priority is set, $reserv$ – the signal to starting a reserve diesel–generator, $lampon$ – the lamp

signalling the diesel–generator 1 started lights up. The requirement part “infinitely often” allow to model cases when hardware failures happen regular, but messages about these failures come with some delays.

SPIN found out a counterexample violated the formula (11) at the depth 29915. The requirement violated because the message which the generator cutout switch 1 controller sent to the power station 1 controller came with delay and blocked starting a reserve diesel–generator.

This error is impossible to detect while bench testing, because it’s impossible to produce unknown quantity of hardware failures with unknown delay. And it’s difficult to detect while program testing because it happens in a very seldom set of circumstances. But because of this error a vessel loses the electrical power control at all.

Interesting that developers observed such a behaviour in vessel sea trials, but they were sure that the error was caused by hardware (not by controllers coordination). So they tried to solve it by adding checks of the generator cutout switch data. And this obviously didn’t help. Developers were not beginners: they specialize in power vessel supply control systems development. Except 23 bench tests they checked the PVSS with 841 program tests. But they didn’t determine the error reason without the requirement formalization and verification.

During verification we detected about 141 errors. Most of them were minor and could be easily fixed, but 3 of them were critical. One of them were discussed above. Second one was about an uncontrollable start of a power station. As result of third critical error hardware could be under the electrical voltage in a PVSS protection mode.

To solve these critical problems it’s required to change controllers algorithms for some modes and add few new modes more. This solution is time consuming, and takes about 80% of the PVSS time design. So we get the well-known consequence that using formal verification methods at first stages of a control program design could allow to avoid subtle, expensive errors at late design stages.

5. Conclusion

We suggested scalable LTL formulas patterns which describe many practical temporal requirements. We show that developing formal temporal requirements with them gives a well-structured requirement specification. The development allows to avoid redundant repeating of temporal requirements and to find out implicit requirements by organizing input-output events and their temporal relations.

Verifying the power vessel supply control system with developed requirements we found out three critical errors. These errors were not found developers by testing. The result of one critical error were observed by developers but they could not determine errors reasons correctly without the requirement formalization and verification. Fixing such critical and subtle errors at late stages of a control program design sometimes could be compared starting a program development from the scratch.

Список литературы

1. Dwyer M. B., Avrunin G. S., Corbett J. C. Patterns in property specifications for finite-state verification // ICSE '99: Proceedings of the 21st international conference on Software engineering. ACM. 1999. P. 411-420
2. Konrad S., Cheng B. H. C. Real-time specification patterns // Proceedings of the 27th international conference on Software engineering. ACM. 2005. P. 372-381
3. Grunske L. Specification patterns for probabilistic quality properties // Proceedings of the 30th international conference on Software engineering. ACM. 2008. P. 31-40
4. Mondragon O. A., Gates A. Q., Roach, S. M. Composite Propositions: Toward Support for Formal Specification of System Properties // Software Engineering Workshop. IEEE. 2002. P. 67-74
5. Mondragon O., Gates A. Q., Roach S. Prospec: Support for Elicitation and Formal Specification of Software Properties // Runtime Verification Workshop. ENTCS. Elsevier. 2004. V. 89. P. 67-88
6. Salamah S., Gates A. Q., Kreinovich V. Validated templates for specification of complex LTL formulas // Journal of Systems and Software. 2012. V. 85. P. 1915-1929
7. Smith R. L., Avrunin G. S., Clarke L. A., Osterweil L. J. Propel: an approach supporting property elucidation // 24th Intl. Conf. on Software Engineering. ACM Press. 2002. P. 11-21
8. Ramezani E., Fahland D., van Dongen B. F., van der Aalst W. M. P. Diagnostic Information for Compliance Checking of Temporal Compliance Requirements // CAiSE. 2013. P. 304-320
9. Post A., Menzel I., Podelski A. Applying restricted english grammar on automotive requirements: does it work? A case study // 17th international working conference on Requirements engineering: foundation for software quality. Springer-Verlag. 2011. V. 6606. P. 166-180
10. Yu J., Manh T. P., Han J., Jin Y., Han Y., Wang J. Pattern Based Property Specification and Verification for Service Composition // 7th International Conference on Web Information Systems Engineering (WISE). Springer-Verlag. 2006. V. 4255. P. 156-168
11. Pnueli A. The temporal logic of programs // 18th Annyv. Symp. on Foundation of Computer Science. IEEE Computer Society. 1977. P. 46-57
12. Karpov Yu.G. Model Checking. Parallel and distributed program systems verification // SPb:BHV-Petersburg. 2010. 560 p. (in Russian)
13. Shoshmina I.V. A method eliciting context requirements to logical control program systems // Information and Control Systems. 2014. №3, P. 68–77 (in Russian)

14. Jackson M. A. Problem Analysis Using Small Problem Frames // South African Computer Journal. 1999. V. 22. P. 47-60
15. Holzmann G. The Spin Model Checker // Primer and Reference Manual Addison Wesley. 2003