Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

A Testing Pattern for Automatic Control Software Addressing Different Degrees of Process Autonomy and Cooperation

Francesca Saglietti, Raimar Lill

Informatik 11 − Software Engineering, University of Erlangen-Nuremberg, Erlangen, Germany
(e-mail: saglietti@informatik.uni-erlangen.de, raimar.lill@informatik.uni-erlangen.de)

Abstract: This article considers different automatic control paradigms allowing for varying degrees of agent cooperation and autonomy. In order to support the automatic verification of safety-relevant software controllers, it proposes the use of a generic testing pattern which can be instantiated such as to allow to optimize automatic test data generation with respect to the specific targets of the application considered and of the testing phase involved. The article reports on successful case studies carried out in different real-world environments.

1. INTRODUCTION

During the last decades, computer-based automatic control has been essentially characterized by increasing demands on software reliability and functional complexity.

On the one hand, the growing number of safety-critical, computer-controlled technical applications in everyday life obviously requires the development of appropriate, transparent and practicable verification techniques for the purpose of ensuring risk-adequate reliability targets.

On the other hand, the positive experiences meanwhile gained with software-based instrumentation and control − allowing in particular to support or even replace costly and fault-prone human resources − motivate the development of new applications based on increasingly sophisticated functionalities. On its turn, the automatic controllability of growingly complex functional behaviour demands for correspondingly more elaborated verification techniques capable of ensuring adequate reliability levels at affordable effort.

Currently, a particular challenge is being posed by the need to control the behaviour of cooperating technical processes typically arising in robot factories or in vehicle traffic based on car-to-car communication. The automatic control of decentralized, concurrent behaviour evidently poses non-obvious communication and synchronization requirements which have to be explicitly addressed during development as well as during verification.

The more complex is the situation in case the cooperating processes to be controlled are planned to act in full autonomy, thus requiring each of the participating agents to be individually computer-controlled. In such a case the controlling software must be specified and designed such as to be able to anticipate any possible interaction with other agents and to enable appropriate decision-making solely on the basis of decentralized sensoring and reasoning capabilities.

Common to all control paradigms summarized in Table 1 is the need to capture the variety of potential scenarios before operation and to verify the corresponding multiplicity of behaviour by appropriate tests. Obviously, this is less difficult in case of one central controller addressing a single technical process than in case of distributed controllers involving interaction among several agents.

Table 1. Control Paradigms

automatic control	*controlled process(es)*
embedded controller, local autonomy	stand-alone process
central controller, no local autonomy	cooperating processes / concurring processes
interacting local controllers, high or full local autonomy	cooperating processes / concurring processes

Depending on the underlying control paradigm, different testing approaches were developed and applied in recent years for the purpose of capturing and covering behavioural variability.

In spite of their differences, they reveal to be based on a generic testing pattern which will be illustrated in this article and which allows to be instantiated depending on the functional complexity and on the application-specific testing targets. This pattern is structured as follows.

1.1 Step 1: Choice of Behavioural Model

Depending on the functionality addressed, a suitable behavioural notation must be selected allowing for the expressive power required by the problem class considered. Higher modelling expressiveness may only be achievable at the price of lower analyzability or even of undecidability. In

Copyright © 2014 IFAC

order to allow for the visualization of scenarios, a graphical notation is usually selected. Some classical examples are illustrated in Chapter 2.

1.2 Step 2: Definition of Testing Targets

Depending on the testing phase addressed (unit testing, integration testing or acceptance testing) the verification process is driven by specific super-ordinate motivations like fault detection, cost or reliability assessment. The degree at which a goal may be considered as adequately achieved has to be captured by suitable metrics addressing a.o. test coverage and test amount. In case the testing targets involve conflicting goals, test data generation results by multi-objective optimization. Some examples are presented in Chapter 3.

1.3 Step 3: Automatic Test Data Generation

In general, complex multi-objective optimization problems do not allow for analytical solutions, but rather require heuristics capable of providing acceptable compromises between full optimization and practicality. Evolutionary techniques based on Darwinian evolution theory have revealed as extremely helpful for the purpose of generating test data sets such as to achieve to an acceptable degree the different targets of a testing phase. Some details will be illustrated in Chapter 4.

2. BEHAVIOURAL MODELLING NOTATION

2.1 Unit Testing of Embedded Controllers

Typical graphical notations modelling the logic of central automatic controllers are characterized by compact visual descriptions of their operational behaviour. At design stage, classical representation languages are *finite state machines* capturing all legal sequences of operational states. At code level, on the other hand, units of limited size are usually represented by *control flow* graphs capturing all legal paths through the software by highlighting the transfer of control between instructions; classical control flow patterns are shown in Fig. 1.

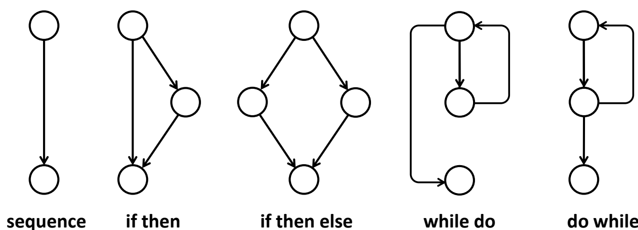

Fig. 1. Control flow graphs modelling the transfer of control between program instructions

Furthermore, control flow models may be enriched by *data flow* annotations indicating at which point during execution variable values are *used* in order to take decisions concerning transfer of control or in order to *define* new values of program variables (*Rapps and Weyuker, 1982*). Evidently, (def,use)-pairs consisting of a variable definition and of any of its uses are suitable to reflect the write-read chains of operations induced by data flow.

2.2 Integration Testing of Component-based Controllers

Once individual components have been thoroughly tested, their appropriate interplay is verified by integration-based test cases focused on *interfaces* and *invocations* across component boundaries.

At design level, such interactions may be modelled by *communicating finite state machines*, each representing the behaviour of one component. Upon invocation of an operation across component boundaries, both the invoking and the invoked component may change their internal state by synchronized traversals of state transitions (*Saglietti and Pinte, 2010*). For example, the occurrence of event tr2 in state1 of component c (Fig. 2) not only triggers the transition of c to state3; it results in a further event tr1' triggering the transition of component c' from state1' to state2' as well.

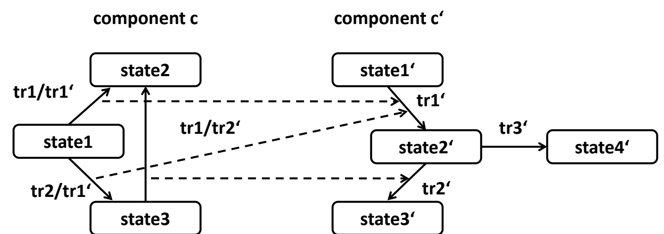

Fig. 2. Communicating finite state machines modelling synchronous component interactions at design level

At code level, crucial interactions between components are captured by applying classical data flow concepts (s. section 2.1) to the data flow occurring across component boundaries (*Jin and Offutt, 1998, Alexander and Offutt, 2004*). An example of *coupling pairs*, i.e. (def,use)-pairs induced by component invocation, is shown in Fig. 3.

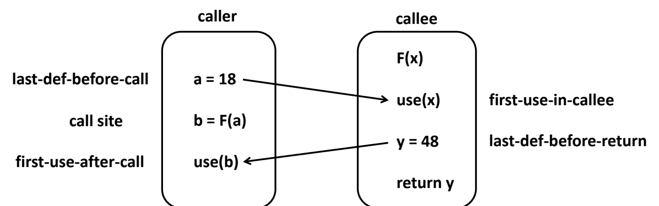

Fig. 3. Coupling pairs induced by component invocation F(a) modelling component interactions at implementation level

2.3 Interaction Testing of Cooperating Controllers

In case of decentralized and interacting controllers the main question concerns the degree of cooperation and autonomy

involved. *Cooperation* implies the coordination of individual contributions required to achieve a common target. In particular, this includes the need of avoiding resp. resolving potential conflicts arising in the course of cooperative activities. On the other hand, *autonomy* − from the ancient Greek terms *auto* (self) and *nomos* (law) − refers to the behavior of entities governing themselves and taking responsibility for their actions. Evidently, in general cooperation may involve different degrees of autonomy, where the most challenging scenario involves fully autonomous agents capable of individual decision-making solely on the basis of their own *perception of reality* and of their *reasoning capabilities* without requiring the intervention of any broker agent suggesting actions on the basis of global knowledge.

A graphical language capable of capturing the interactions of autonomously cooperating processes was shown to be offered by *coloured Petri nets* (*Jensen and Kristensen, 2009*). In general, classical *Petri nets* allow to generalize from *finite state machines* by allowing the current state information to be spread across the whole graph by means of tokens, hereby stressing the decentral role of concurrent processes. In particular, *Petri nets* also permit to represent infinitely many states by means of finite graphs. In case of varying numbers of cooperating agents, however, the choice of classical *Petri nets* would require to re-design the cooperation model each time new agents are introduced, e.g. as soon as a new intelligent forklift is added to a robot factory. The alternative offered by *coloured Petri nets* (CPNs) allows to relax this restriction by including several types (*colours*) of tokens for the purpose of encoding several robot missions, thus enabling high model scalability, as a fluctuating number of agents is easily captured by adding or removing tokens without needing to adapt the underlying graphical structure.

A *coloured Petri net* illustrating the cooperation of forklifts moving along a path consisting of discrete segments is illustrated in Fig. 5. A central controller assigns dedicated missions to the forklifts (transition *next order*). Each robot aims at achieving its own task as autonomously as possible by accessing segments previously identified as free (transition *forward* resp. transition *backward* depending on the direction taken). In case forklifts face each other while trying to reach their target, they decide to exchange their positions (transition *switching maneuver*). Segments may be blocked by traffic jam or by passive obstacles; after 5 unsuccessful attempts to access a segment, an alarm is raised (transition *traffic holdup*). Once a robot reaches its target, the successful completion of its mission is eventually logged (transition *mission completed*). Thanks to its high scalability, the same CPN layout allows to capture the behaviour of an arbitrary number of forklifts moving along an arbitrarily long path by adjusting the number of token colours accordingly.

The different behavioural models addressed in this article are summarized in Table 2.

Table 2. Behavioural Models

automatic control software	***graphical modelling notation***
central controller, monolithic code	finite state machine, control flow graph, data flow annotations
central controller, component-based code, synchronous execution	interface / invocation model, communicating state machines
decentral controllers, cooperating codes, asynchronous execution	Petri net, coloured Petri net

3. MEASURABLE TESTING TARGETS

As already mentioned, each testing phase is driven by phase-specific targets and constraints, a.o. behavioural coverage, test case balance, prioritization of test scenarios, operational significance and cost limits.

3.1 Fault Detection by Structural Coverage

Early testing phases are mainly characterized by the need to detect faults. Therefore, test criteria applied to these phases usually refer to measures of behavioural coverage to be maximized during testing. Such measures are expected to correlate with fault detection capability, as faults affecting regions not covered by test cases are doomed to remain undetected.

Depending on the models introduced in Chapter 2, behavioural coverage can be measured by the relative amount of graphical elements (*nodes*, *arcs* and *paths*) traversed during testing. Several model-based testing criteria were defined for the models introduced above, a.o. *statement*, *branch* and *path* coverage for unit control flow, several variants of (*def,use*)-*pairs* coverage for unit data flow (*Rapps and Weyuker, 1982*), *coupling pairs* coverage and *pairs of interacting transitions* for component integration (*Jin and Offutt, 1998, Saglietti and Pinte, 2010*), as well as various coverage criteria for coloured Petri nets (*Lill and Saglietti, 2013*). The latter are based on the coverage of CPN *transitions* (representing generic, context-free actions), CPN *events* (data-enriched transitions representing context-dependent actions), up to CPN *states* (token markings globally capturing any operational condition which may be encountered before or after the traversal of a transition).

In addition to coverage demands, test cases may also be required to be spread as evenly as possible over all regions of the subject under test, hereby aiming at a well-balanced test case distribution. Alternatively, test intensity may be varied according to the safety relevance of the functionalities addressed, possibly yielding manifold coverage demands for more critical regions as opposed to simple coverage demands for behavioural scenarios without safety implications.

3.2 Testing Effort due to Test Amount

For practical reasons, the testing effort should be kept as low as reasonably possible; in particular, the number of test cases required to achieve a given coverage should be minimized by avoiding redundant testing evidence, as each test case demands for an accurate validation of the test behaviour observed.

3.3 Reliability Analysis by Independent Operational Sample

For the purpose of deriving a quantitative assessment of software reliability during the final acceptance testing phase, a sound technique called *reliability testing* consists of observing an operationally significant sample of test scenarios and to infer from their correctness an upper bound of software failure probability to any given confidence level α. The conservative estimation is based on *statistical sampling theory* (*Parnas, Schouwen and Kwan, 1990*).

Its application requires that the n scenarios observed are significant for the expected operational behaviour; in particular, this means that the observations are not redundant and that they reflect an application-specific usage profile. In other words, the test cases have to be independently selected according to an expected operational profile. Furthermore, they are also required to be independently executed, which, if required, can be achieved by resetting mechanisms between subsequent test case executions. Under these conditions, reliability testing without failure occurrence allows to bound software failure probability p at confidence level α by

$$p \leq 1 - \sqrt[n]{1-\alpha}\ .$$

With regard to test case selection, the main testing criteria concern conformity to a given profile and stochastic independence. Classical goodness-of-fit-tests like the χ^2*-test*, the *Kolmogorov-Smirnov test* or the *Anderson-Darling test* may be applied to evaluate whether test data can be taken as sufficiently representative for a given operational distribution. On the other hand, stochastic independence among test scenarios can be measured by evaluating appropriate *auto*- and *cross-correlation* coefficients.

Test drivers with corresponding measurable criteria are summarized in Table 3.

Table 3. Testing Targets

test driver	*criterion*
fault detection	structural coverage
balance	evenness of coverage
safety	prioritization according to criticality
effort	amount of test cases
reliability	operational significance (representativeness, independence)
representativeness	goodness-of-fit tests
independence	auto-and cross-correlation tests

3.4 Multi-Objective Optimization

In practice, depending on the testing phase an appropriate combination of targets is considered. Typical examples of multiple objectives to be pursued in the same testing phase were already mentioned in earlier sections and summarized in Table 4.

The simultaneous achievement of some of these goals may involve high complexity, especially in case of conflicting objectives as those involved when maximizing structural coverage while minimizing the number of test cases required.

Evidently, in general such multi-objective optimization problems cannot be solved analytically, but rather demand for heuristic approaches, as presented in the next Chapter 4.

Table 4. Multi-Objective Testing

testing phase(s)	*objectives*
unit test, integration test, interaction test	maximization of structural coverage minimization of number of test cases
reliability test	operational representativeness stochastic independence
reliability test combined with integration test	operational representativeness stochastic independence maximization of interaction coverage *additional optional objectives:* evenness of coverage or prioritization according to criticality

4. AUTOMATIC DATA GENERATION BY HEURISTICS

The heuristic technique applied to generate optimal test data with respect to different − sometimes conflicting − objectives is based on Darwinian evolution theory: each individual belonging to an initial random population is evaluated in terms of the degree to which it achieves the underlying targets; the result of this evaluation is a quantitative measurement of its fitness. As long as no individual can be taken as acceptable, a new population is derived by the preceding one by means of genetic manipulations and the evaluation is repeated. The iterative process is illustrated in Fig. 4, while the genetic operators and the correspondence between genetic entities and test data structure are detailed in Table 5 and in Table 6.

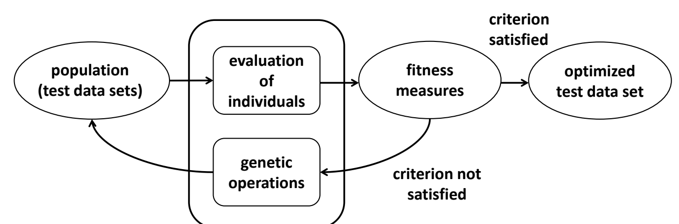

Fig. 4. Evolutionary process applied to test data generation

Table 5. Genetic Operators and Strategies

genetic operator / genetic strategy	*effect*
selection	extraction of best fitted individuals or random extraction of individuals
recombination by uniform crossover	exchange of genes between individuals at given probability (possibly decreasing with progress)
recombination by cut & glue	split of 2 genes of 2 individuals, recombination after switching parts
mutation	delete, add, replace, modify genes
gene pool	storage of good genetic material to be re-inserted if required
skip crossover	bypassing of recombination to avoid worsening best individuals
elitism	unaltered transfer of fixed amount of best individuals

Table 6. Analogy between Genetics and Testing

genetic entity	*test element*
gene	test case: initial state followed by sequence of parameterized actions
individual	test suite: set of test cases, output of test case generator
population	set of test suites, candidates for test case generator

The fitness of an individual with respect to several objectives may be simply evaluated by weighted combinations of target-specific fitness values; this requires, however, a priori decision-making on weight values. Alternatively, Pareto-optimal solutions (not dominated by any other solution with respect to all objectives) may be heuristically determined and visualized by a Pareto front. This strategy supports a posteriori decision-making by moving along the front and selecting acceptable solutions in the light of individual cost-benefit considerations.

5. RESULTS

The testing pattern proposed in this article was instantiated via implementation of tools successfully applied to several application classes.

At the level of unit testing, first experiences were gained with the maximization of control and data flow coverage by a minimal number of test cases (*Oster and Saglietti, 2006*).

At the level of integration testing, an approach to capture interactions by synchronous communication between software components was developed and applied in a medical environment (*Saglietti and Pinte, 2010*).

At the level of reliability testing, automatic test case evaluation and generation approaches were developed and applied to a software-based gearbox controller in order to extract significant runs from operating experience gained during test drives and to support the extension of these operational runs by additional, both independent and representative test cases allowing for reliability assessment (*Söhnlein et al., 2010*).

Furthermore, reliability testing was combined with integration testing by requiring – in addition to operational significance − also the maximization of coupling pair coverage. This helps ensuring that the operational evidence on which to base reliability assessment is not biased by omitting less frequent component interactions. This approach was further extended to include test evenness over interactions and prioritization of critical interactions (*Meitner and Saglietti, 2014*).

Finally, for the purpose of testing the interaction of autonomous cooperating agents, the approach proposed was applied to a model of cooperative forklifts inspired by a real-world robot logistic warehouse (*Lill and Saglietti, 2013*).

REFERENCES

Alexander, R.T., Offutt, A.J. (2004). Coupling-based Testing of O-O Programs. In: *Journal of Universal Computer Science*, vol. 10(4). TU Graz

Jensen, K., Kristensen, L.M. (2009). *Coloured Petri Nets − Modelling and Validation of Concurrent Systems*. Springer-Verlag

Jin, Z., Offutt, A.J. (1998). Coupling-based Criteria for Integration Testing. In: *Software Testing, Verification & Reliability*, vol. 8(3). John Wiley & Sons

Lill, R., Saglietti, F. (2013). Model-based Testing of Cooperating Robotic Systems using Coloured Petri Nets. In: *Proc. SAFECOMP 2013 Workshops*. LAAS-CNRS and open access archive HAL

Meitner, M., Saglietti, F. (2014). Target-Specific Adaptations of Coupling-Based Software Reliability Testing. In: *Measurement, Modelling and Evaluation of Computing Systems, Dependability and Fault Tolerance*, Lecture Notes in Computer Science, vol. LNCS 8376. Springer-Verlag

Oster, N., Saglietti, F. (2006). Automatic Test Data Generation by Multi-Objective Optimisation. In: *Computer Safety, Reliability and Security*, Lecture Notes in Computer Science, vol. LNCS 4166. Springer-Verlag

Parnas, D., van Schouwen, J., Kwan, S. (1990). Evaluation of Safety-critical Software. In: *Communications of the ACM*, vol. 33 (6). ACM

Rapps, S., Weyuker, E.J. (1982). Data Flow Analysis Techniques for Test Data Selection. In: *6th Int. Conf. on Software Engineering*. IEEE Computer Society

Saglietti, F., Pinte, F. (2010). Automated Unit and Integration Testing for Component-based Software Systems. In: *Dependability and Security for Ressource Constrained Embedded Systems*. ACM Digital Library

Söhnlein, S., Saglietti, F., Bitzer, F., Meitner, M., Baryschew, S. (2010). Software Reliability Assessment Based on the Evaluation of Operational Experience. In: *Measurement, Modelling, Evaluation of Computing Systems, Dependability and Fault Tolerance*, Lecture Notes in Computer Science, vol. LNCS 5987. Springer-Verlag

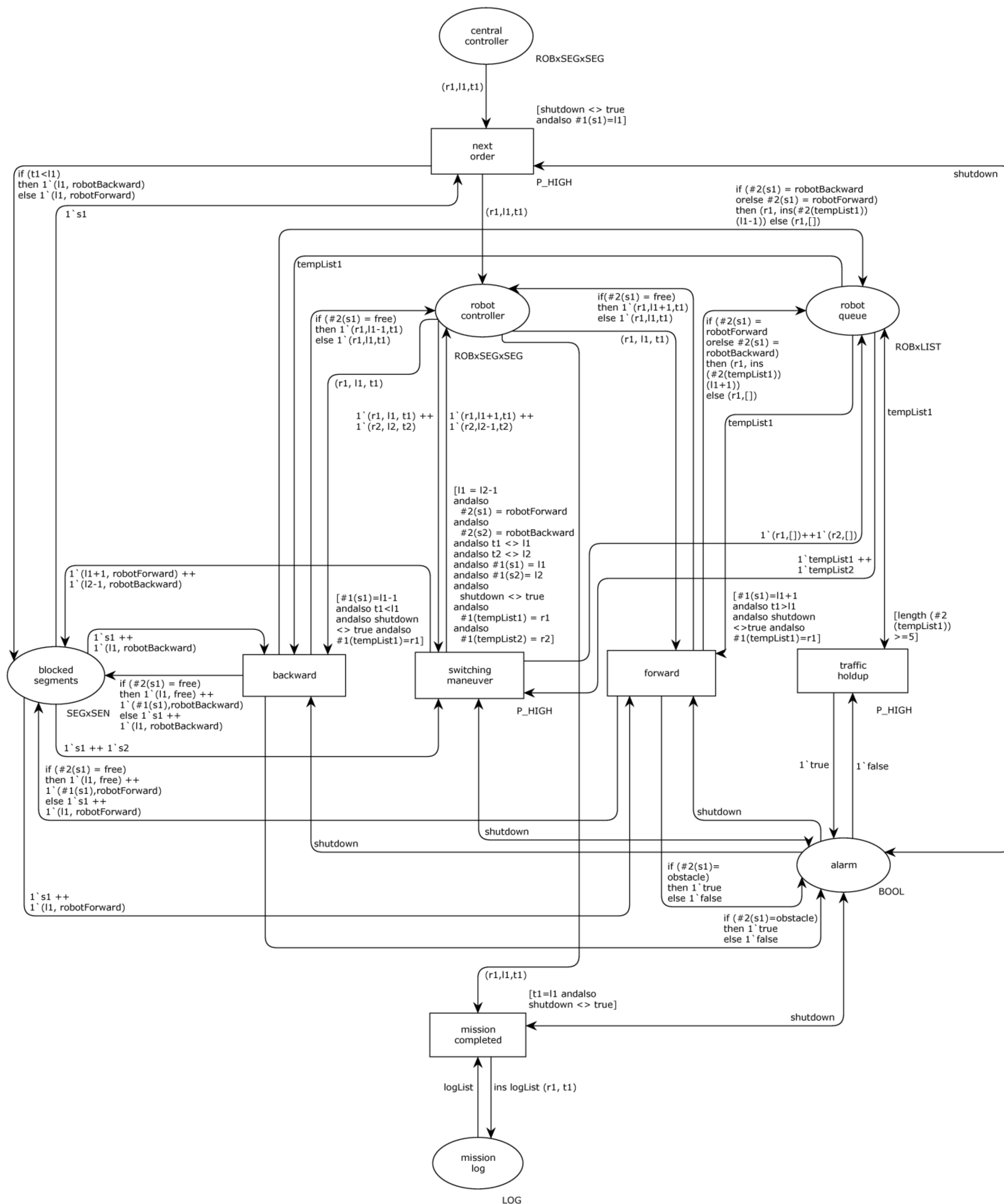

Fig. 5. Coloured Petri net modelling the cooperation of autonomous forklifts moving along a common path