УДК 519.172

Verification of UCM Models with Scenario Control Structures Using Coloured Petri Nets

Vizovitin N.V. (A.P. Ershov Institute of Informatics Systems SB RAS),
Nepomniaschy V.A. (A.P. Ershov Institute of Informatics Systems SB RAS, Novosibirsk State University)
Stenenko A.A. (A.P. Ershov Institute of Informatics Systems SB RAS)

This article presents a method for the analysis and verification of Use Case Maps (UCM) models with scenario control structures – protected components and failure handling constructs. UCM models are analyzed and verified with the help of coloured Petri nets (CPN) and the SPIN model checker. An algorithm for translating UCM scenario control structures into CPN is described. The presented algorithm and the verification process are illustrated by the case study of a network protocol.

Keywords: *verification, translation, Use Case Maps notation, coloured Petri net, SPIN model checker, protected component, failure handling.*

1. Introduction

At early stages of software projects development during requirements capturing and analysis error prevention is of importance due to high cost on this stage. Use Case Maps (UCM) scenario-oriented graphical notation [10] allows users to formalize and analyze functional requirements. At the same time, it allows customers to monitor the system requirements. UCM models are general purpose. They are used for test case generation [3, 4], building test coverage criteria [2], and as a property specification language [8] for use with model checkers.

UCM model of a system depicts a set of scenarios as cause-and-effect relations between responsibilities. Responsibilities may be superimposed on the underlying components structure, reflecting the architecture of the system. UCM describes interaction of architectural entities focusing on causal relations and abstracting from some details of messaging and data processing.

However, tools for analysis and verification of UCM models are insufficiently developed. The UCM standard [10] defines an analysis procedure, which is implemented in the jUCMNav editor [11]. This analysis technique is rather primitive and it is hard to use. Since the standard describes the language semantics informally using traversal requirements for UCM, a number of papers are focused on providing a formal UCM semantics [6]. A few papers present a solution for verification of UCM models [7]. Verification methods for specific subject domains are also being developed. The paper [1] describes testing, analysis, and verification methods for telecommunication applications based on UCM models.

In previous papers [15, 16], we described an approach for general-purpose UCM models analysis and verification using coloured Petri nets (CPN). UCM models are translated into CPN models. The latter are then verified using the well-known SPIN model checker [9]. CPN models may also be analyzed directly using CPN Tools [5].

This article extends the scope of previously supported UCM constructs with protected components and failure handling constructs. It describes a method of analysis and verification for these UCM constructs.

2. Use Case Maps Notation Overview

The Use Case Maps notation is one of the languages defined in the User Requirements Notation standard [10]. The UCM visual notation is a high-level scenario-oriented modeling tool. It focuses on the causal flow of behavior, which is optionally superimposed on a structure of components. UCM models depict the causal interaction of architectural entities in a system while abstracting from message passing and data details. The notation simplifies modeling and analysis of functional requirements for distributed and concurrent systems while also allowing to reason about system architecture.

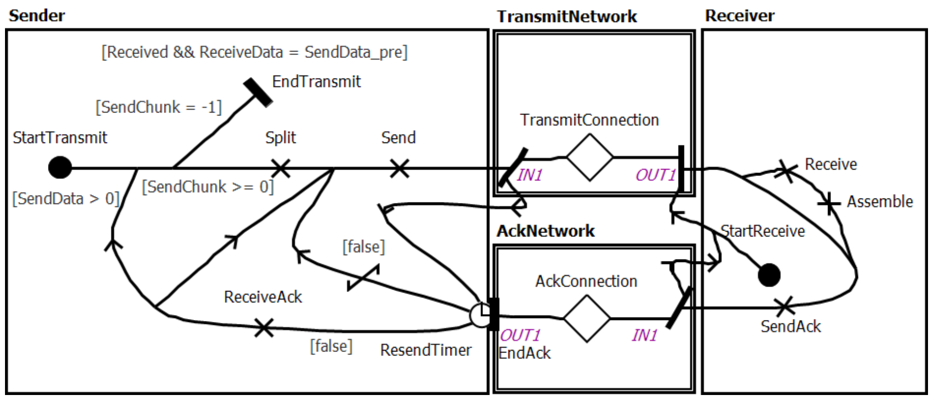

Fig. 1. Top-level map of the network protocol UCM model

Below we provide a short overview of basic elements of the UCM notation. Detailed language description including its graphical syntax is provided in [10] and [15]. A *map* (see Figure 1) contains any number of paths and components. *Paths* (depicted as connecting lines) express causal sequences and causal relationships between path nodes. Paths are directed. They may contain several types of path nodes. Paths start at *Start Points* (for example, `StartTransmit` on Figure

1) and end at *End Points* (`EndTransmit`). These nodes define triggering and resulting conditions respectively or pre-conditions and post-conditions (shown in square brackets). Start Points also may denote the beginning of scenarios for failure and exception handling. Such Start Points are called *Failure Start Points* and *Abort Start Points* respectively. Start Point type is defined by a `failureKind` attribute, while a `failureList` attribute specifies a list of failures it may respond to. Abort Start Point is a Failure Start Point that in addition cancels all scenario behaviors in its *abort scope* – map of the Abort Start Point as well as all lower level maps as defined by Stub hierarchy (see below). *Responsibilities* (`Split`) define steps or actions required to fulfill a scenario. *Or-Forks*, possibly including conditions for outgoing path selection (shown in square brackets), and *Or-Joins* are used to model alternatives and loops. *And-Forks* and *And-Joins* express concurrency. *Waiting Places* and *Timers* (`ResendTimer`) denote points on the path where a scenario stops until a condition is satisfied or a triggering signal arrives. Scenario may also continue past the Timer using the timeout path. *Connect* nodes and *Empty Points* are used to connect two paths synchronously or asynchronously. *Failure Points* represent points on a path where the continuation of a scenario depends on the occurrence of a failure or exception. Each failure point has an associated triggering condition, as well as a failure name, which indicates the failure or exception that happened. Failure name effectively defines Failure or Abort Start Points used to continue scenario execution in case triggering condition is true. UCM models can be hierarchically decomposed using *Stubs* (`TransmitConnection`) that contain reusable units of behavior and structure called *plug-in maps*.

Components (`Sender`) are used to specify structural aspects of a system. Path nodes that reside inside a component are said to be *bound* to it. UCM models without components are said to be *unbounded*. Components may contain sub-components and have various types. However, most of them do not influence model semantics and serve only to convey architectural aspects of a system. Exceptions include components of kind *Object* that force interleaved traversal of path nodes of parallel branches that are bound to the component and *protected* components (`TransmitNetwork`) that restrict the amount of concurrent scenarios inside them. In the URN standard, maximum amount of concurrent scenarios inside a protected component is always 1. Therefore, protected components work as a mutual exclusion mechanism for concurrent scenario execution.

3. UCM Models to Coloured Petri Nets Translation Method

To analyze and verify UCM models we translate them into coloured Petri nets [12]. Input and output models are represented as hierarchical directed graphs with additional information associated with vertices and arcs.

3.1. Input UCM Model Restrictions

The following important restrictions are imposed on the input UCM models. All model elements have unique names and direction of all paths is defined. Special traversal semantics for the components of type Object is not supported. UCM models with Abort Start Points are rejected. These path nodes are rarely used and cause state space explosion upon translation due to scenarios termination semantics. Therefore, we do not translate them to CPN.

For the translation of protected components, we also impose an additional restriction on Or-Fork, Timer, and Or-Join nodes. Each path node of these types should be either bound or not to a given protected component together with all of its adjacent path nodes. This limitation is not fundamental since it could be achieved by simple UCM model modification. In case the system detects that this limitation is not held, the user is offered to modify the UCM model by either introducing additional Empty Point nodes adjacent to the problematic path nodes or in any other way that ensures that the limitation is held.

The listed restrictions do not limit significantly the set of supported UCM models since the most used elements and their use cases are supported.

3.2. UCM to CPN Translation Algorithm Overview

On the top level, UCM to CPN translation algorithm consists of five steps. On the first step pre-processing of an input UCM model is performed. The first step includes simple conversions of an input model as well as checks of input model constraints. The second step creates various CPN ML language definitions common to the entire CPN model. On the third step, additional vertices are added to the UCM model graph to simplify its conversion to a bipartite graph. On the fourth step, path node vertices with their immediate vicinity are translated independently of each other according to their types. The fifth step combines CPN fragments produced on the previous steps into a single CPN model. The translation algorithm is described in detail in [15, 16].

On the first step, UCM model is pre-processed. As part of this process the model is converted to an unbounded one, i.e. all components are removed. Information about protected components is stored in the attributes of each path node bound to the given protected component. All initial values for variables in the UCM model are determined. Algorithm constraints for input models are checked. The system notifies the user about any implicit conversions during this step.

The second step defines colours, constants, and some variables. A number of auxiliary colours are introduced, including `UNIT` – standard “base” colour with only one possible value `()`. Tokens with the colour `UNIT` are normally used to model signal transmission or scenarios execution.

On the third step, graph arcs that are not incident to vertices representing Connect path nodes are partitioned. Each arc is partitioned into two arcs using new helper vertices i.e. vertices of the new type *FakePathNode*. The resulting arcs preserve directions as well as annotations on the arcs outgoing from non-helper vertices.

On the fourth step, each path node vertex with its immediate neighborhood defined by adjacent Connect and helper vertices is handled separately. Each path node and its neighborhood are translated into a CPN model fragment – an annotated graph with additional definitions in CPN ML language. For each failure name, a CPN model fragment is also generated. During translation, helper vertices become places of type UNIT.

The fifth step combines CPN model fragments produced on the fourth step into a single resulting CPN model. Model elements with same names are either merged or represented as fusion places if necessary.

4. Translation of Path Nodes Bound to Protected Components

To verify UCM models efficiently using CPN, number of scenarios being executed at a given point of the model should be limited. Otherwise, translated CPN model will have places with unbounded place capacity since places are used to model signal transport.

The UCM standard provides a method for modeling mutual scenarios exclusion for a subset of the UCM model paths. Protected components depicted with a double outline fulfill this purpose. All UCM model path nodes bound to the protected component are affected by it. Execution of any scenario may continue inside a protected component only if no other scenario is already being executed inside of it.

However, the semantics of the protected components offered by the standard is too restrictive to represent a wide variety of scenarios interactions while keeping the capacity of CPN places in the translated model limited. Thus, we propose to extend the standard by allowing to specify a maximum amount of concurrent scenarios within a protected component. This could be implemented either by adding a new integer attribute `scenarios` into the *Component* class of UCM abstract grammar or by using comment elements attached to a given protected component. The latter approach may be used to avoid modifying existing UCM editors.

Below we will describe translation of UCM components with attribute `protected = true` and positive values of the `scenarios` attribute, as well as other UCM path nodes which are bound to such components. Note that if `scenarios = 1` then the protected component has the same semantics as in the UCM standard.

As any other UCM element, protected components have unique names. We will also impose an additional restriction for Or-Fork and Or-Join nodes. Each path node of these types should be either bound or not to a given protected component together with all of its adjacent path nodes. This limitation is not fundamental since it could be achieved by a simple UCM model modification. However, it avoids semantics ambiguity for such UCM models, as well as significant complication of the translation algorithm.

The example of protected components translation from Figure 1 is provided in Section 7 and a more detailed description of it is given in [14].

On the first step of the translation algorithm, we additionally check that `scenarios > 0` for all protected components. Otherwise, UCM model is deemed incorrect. The additional limitation on Or-Fork and Or-Join nodes is checked as well. If it does not hold user is advised to modify the UCM model by adding new Empty Points on the arcs incident to the problematic path nodes and adjusting protected components. Information about each protected component is stored in the attributes of path nodes bound to it. Each path node may be bound to multiple protected components.

The second and third steps of the translation algorithm have nothing specific for protected components. They are considered in Section 3.2.

Protected components are modeled in CPN using anti-places. Anti-place is a common CPN modeling pattern used to limit the amount of tokens in a given fragment of CPN. Initial marking of an anti-place usually holds the amount of UNIT tokens equal to the limit. When other tokens are created in a given CPN fragment an equal amount of tokens from the anti-place should be consumed. When other tokens are removed from a given CPN fragment, an equal amount of tokens should be put back to the anti-place.

On the fourth step, each path node vertex and its adjacent vertices is translated into a CPN fragment. An anti-place is created for each protected component a vertex has information about in its attributes. The anti-place has a colour UNIT and an initial marking with the same amount of tokens as the value of the `scenarios` attribute was for the protected component. Only the nodes that are capable of starting (forking) or terminating (joining) scenarios that flow through them and a protected component will actually create additional anti-places and arcs. Anti-places are named after the corresponding protected components, so they are uniquely identifiable as well.

The fifth step of the translation algorithm stays the same – all additional anti-places will be joined according to their names in the same way other places of CPN fragments are joined, using fusion places if necessary.

Translation of separate UCM path nodes is described below. For each path node and each protected component, we may define whether this path node and any of the path nodes adjacent to it are bound to the component. For Or-Fork and Or-Join nodes the path node itself as well as other path nodes adjacent to it are either all bound or not to a given protected component. They do not create or terminate scenarios. Therefore, CPN transitions corresponding to Or-Fork and Or-Join nodes never need to be connected to anti-places.

Let us consider the translation of path nodes that may create or terminate scenarios. These include And-Forks, And-Joins, Start Points, and End Points. If a path node is bound to a protected component, a difference between the number of outgoing and incoming arcs is calculated. In case it is positive, an arc is added to the resulting CPN fragment from the anti-place to the transition corresponding to the path node. In case it is negative, an arc is added in the reverse direction. In both cases, arc inscription equals to the () times the absolute difference value. Note that difference value cannot be zero – otherwise, there is no scenario creation or termination.

Let us consider the translation of path nodes bound to a protected component that have adjacent path nodes not bound to the component. We calculate a balance value for a path node. Starting balance value is 0. Each outgoing arc that leads to a path node not bound to the component decreases balance by 1. Each incoming arc from a path node not bound to the component increases balance by 1. After considering all incident arcs for the given path node we have a balance value of this path node in relation to the protected component. In case the balance value is positive, an arc is added to the resulting CPN fragment from the anti-place to the transition corresponding to the path node. In case it is negative, an arc is added in the reverse direction. In both cases, arc inscription equals to the () times the absolute balance value. In case the balance is zero no new arcs are added.

The additional arcs described above may be added independently of one another. In this case, their inscriptions are combined in a natural way. If a stub is bound to a protected component then the described procedure is applied to all path nodes on child diagrams of the stub as well, accounting for Start Points and End Points that have bindings to the stub, which do not create or terminate scenarios.

5. Translation of Failure Handling Path Nodes

Failure handling in UCM is modeled using Failure Point and Failure Start Point path nodes. Failure Point represents a point in scenario behavior where the continuation of the scenario depends on the occurrence of failure or exception. Failure Start Point denotes the beginning of a scenario behavior in response to failure or, in other words, a start of a failure handler.

Failure Start Point nodes have a list of failure names they are supposed to be triggered for, while Failure Point nodes have a `failure` attribute that denotes the name of failure that is triggered. After a failure is triggered, the triggering scenario at the Failure Point terminates, and a number of scenarios start at Failure Start Point nodes with a matching failure name in their failure list.

On the fourth step of the translation algorithm, each failure name is translated into a CPN model fragment – a transition and a place named after the failure name. The place has `UNIT` type with empty initial marking. An arc leading from the place to the transition is added with a `()` inscription. Arcs are also added from the transition to each place matching Failure Start Points with a corresponding failure name in their failure list. Each of these arcs has a `()` inscription as well. This translation procedure closely resembles And-Fork path nodes translation [15].

Translation of Failure Start Points is similar to ordinary Start Points on child maps when they are bound to stub inputs [16]. Transition corresponding to a Failure Start Point is linked with a place of type `UNIT`, with empty initial marking. Arc from the place to the transition has a `()` inscription.

Translation of a Failure Point is similar to an Or-Fork [15], which has two output paths – one that continues the normal execution of the scenario and one that leads to a placed named after the failure name. The conditions on the output paths are based on the failure condition – one of them is the failure condition and the other one is its negation. Therefore, there is no need for additional `*_OrForkWarnings` place which normally tracks that conditions on the output paths of an Or-Fork path node are mutually exclusive.

Note that a Failure Point and Failure Start Points it triggers may be on different maps. This case is automatically resolved on the fifth step of the algorithm by converting some of the places adjacent to the transition that corresponds to the triggered failure name into fusion places when joining CPN model fragments.

6. Verification of CPN Models

A CPN model translated from UCM model may be analyzed using CPN Tools [5, 12] facilities. In fact, it is especially useful for simulation. It also provides some limited state space analysis tools. However, we find that certain model properties may also be formally verified in an automated and more efficient way. We use our own verification system for CPN that uses well-known SPIN model checker [9]. In order to employ SPIN, CPN models are translated into its input language Promela [13].

Properties for verification are expressed either as simple predicates that are expected to be true at the end state (any state without enabled transitions) or as linear temporal logic formulas. In the former case, the property check is represented as an assertion at the end states of the Promela model. In a number of cases, properties for verification may be derived from the UCM model itself. A common choice is to verify that End Points post-conditions hold and the UCM model is correct with respect to branching conditions. Since UCM models are translated into CPN in a way that provides auxiliary warning places to track the branching errors, it is possible to define such kind of property for verification. On the CPN model level, the property holds if all warning places are empty and all post-condition places contain only `true` tokens in the end state. This is translated to Promela model level as a conjunction of several simple state conditions, which is asserted for the end states.

Several restrictions on the input CPN models are imposed to translate them into Promela language. CPN models produced from UCM models by our translation algorithm conform to all of these restrictions but the finiteness restriction required to verify a model efficiently. CPN models are expected to be finite – places and data types' capacities should be limited. The finiteness restriction may be viewed as a reflection of real computer memory finiteness. It is possible to set all finiteness limits manually before the verification.

The finiteness restriction may be conformed to in various ways – by either constructing a UCM model in a certain way or applying additional restrictions on the Promela model level for state space exploration. In case a given finiteness limit is reached during a verification the system advises the user to either increase the limit value or modify the UCM model by adding protected components to it. Protected components are used as a means to limit places capacity. At the same time, protected components usually identify an important limitation on the UCM model level, such as a limited network bandwidth or a limited amount of memory available to the system.

Verification may be successful or not. If the given property does not hold, a counterexample is generated. A counterexample is a sequence of states (places with their markings) and binding elements (transitions and their variable bindings) that lead to the found invalid state or does not satisfy linear temporal logic formula if the property was specified as one. For user convenience, counterexamples may then be mapped back to the UCM model or analyzed with CPN Tools. After correcting issues in either the UCM model or the property to verify, the verification process is repeated.

7. Case Study

We demonstrate algorithms and tools presented in this paper in a case study. A UCM model describes a simple communication protocol designed to transfer reliably a decimal number over a network capable of transmitting only one digit per packet. Data to transfer is integer due to URN Data Language limitations. The UCM model is translated to CPN and then to Promela model, which is then executed to verify a post-condition from the UCM model. The case study demonstrates usage of protected components to ensure the translated model is finite.

Figure 1 shows a top-level map of the UCM model of the protocol. All UCM elements except fork and join elements are labeled. URN Data Language expressions (branch conditions and post-conditions) are depicted as labels with code in square brackets. URN Data Language actions (associated with Responsibilities depicted as crosses) and some expressions are not shown. The model includes four components: Sender, Receiver, TransmitNetwork, and AckNetwork. The latter two components are protected and were added by the user after the initial verification attempt failed. These protected components limit the amount of concurrent scenarios to 2 and reflect a limited network bandwidth. Sender splits `SendData` value into digits and sends them over the network, retransmitting as necessary. Each of the two network components contains a Static Stub that represents an unreliable network environment for transmitting packets from Sender to Receiver and vice versa. Both stubs contain the same Connection plug-in map. Receiver processes packets as soon as they arrive and assembles transmitted data from them. Receiver acknowledges each arriving packet with a sequence number of the next expected packet. Sender receives acknowledgement packets and updates the sequence number of the next packet to send. Sender assumes that sequence numbers can only increase.

After sending a packet, Sender waits on a Timer element. If the current packet sequence number equals to the sequence number of the next packet to send, then the same packet is resent. Otherwise, Sender fetches the next digit to send and sends a new packet with the next sequence number. A packet with the payload `-1` signals the end of data. If Sender receives an acknowledgement that such packet was received, data is considered transmitted and the `EndTransmit` End Point post-condition `[Received && ReceiveData = SendData_pre]` is checked, where `SendData_pre` is the initial value of the `SendData` variable. The post-condition is satisfied if Receiver considers the data received (an appropriate flag is `true`) and the data received equals to the data sent.

The UCM model post-condition for the `EndTransmit` End Point is verified, together with the absence of warnings during model execution. According to this property, the protocol always finishes in the expected correct state and the source UCM model is consistent with respect to branching conditions. The property is simply asserted in the resulting Promela model at end states. The UCM model, CPN and Promela intermediate models with verification results and a detailed description can be found in [14]. During verification, no additional restrictions were imposed on the Promela model. Verification was successful.

8. Conclusion

The Use Case Maps graphical notation provides an expressive means of describing functional requirements for software systems and protocols. In this article, we have presented a method for translation of UCM models to CPN and its application for verification of UCM models. This method enables users to analyze and verify more expressive UCM models as compared with the method in [15, 16] by supporting failure handling and protected components.

Protected components with the extended semantics are especially useful for verification. Protected components limit the number of concurrent scenarios thus limiting places capacity in the translated CPN model. This ensures that the model is finite and can be efficiently verified using SPIN.

A current version of our tool supports translation of jUCMNav editor [11] files to CPN Tools [5] files. UCM models translated to CPN can be analyzed using either built-in CPN Tools facilities or the CPN models verifier based on SPIN [13]. A verification result shows if a model is correct with respect to a given property. If not, an error must be located. While it is possible to map the counterexample generated by SPIN to the UCM model, we find that it is often more convenient and productive to perform the required analysis using CPN Tools.

The algorithm for UCM models translation into CPN is efficient. The translation method described in [15, 16] has polynomial complexity for the size of the resulting CPN models [16]. This estimate holds for the translation algorithm described in this paper as well.

It is important to justify that UCM to CPN translation is correct. However, this requires a formal semantics for the UCM, which is not provided by the standard [10].

We plan to evaluate our tools using other UCM models of communication protocols as well as other systems. We also plan to explore timing extensions [7] for the UCM notation.

9. References

1. Anureev I., Baranov S., Beloglazov D., Bodin E., Drobintsev P., Kolchin A., Kotlyarov V., Letichevsky A., Letichevsky A. Jr., Nepomniaschy V., Nikiforov I., Potienko S., Pryima L., Tyutin B.

Tools for Supporting Integrated Technology of Analysis and Verification of Specifications for Telecommunication Applications // SPIIRAN №1.- St.Petersburg, 2013 (in Russian).

2. Baranov S., Kotlyarov V., Weigert T. Verifiable Coverage Criteria for Automated Testing. // SDL 2011, LNCS 7083.- 2011.- P. 79-89.
3. Baranov S.N., Drobintsev P.D., Kotlyarov V.P., Letichevsky A.A. The Technology of Automated Verification and Testing in Industrial Projects. // Proc. IEEE Russia Northwest Section, 110 Anniversary of Radio Invention Conference.- IEEE Press, St.Petersburg, 2005.- P. 81-89.
4. Boulet P., Amyot D., Stepien B. Towards the Generation of Tests in the Test Description Language from Use Case Map Models. // SDL 2015, LNCS 9369.- Springer.- 2015.- P. 193-201.
5. CPN Tools Homepage, `http://cpntools.org/`
6. Hassine J., Rilling J., Dssouli R. Abstract Operational Semantics for Use Case Maps. // FORTE 2005, LNCS 3731.- Springer.- 2005.- P. 366-380.
7. Hassine J. Early modeling and validation of timed system requirements using Timed Use Case Maps. // Requirements Engineering v. 20 №2.- 2015.- P. 181-211.
8. Hassine J., Rilling J., Dssouli R. Use Case Maps as a Property Specification Language. // Software and Systems Modeling 8(2).- 2009.- P. 205-220.
9. Holzmann, G.J.: The SPIN model checker. Primer and Reference Manual.- Addison-Wesley, 2004.
10. ITU-T, Recommendation Z.151 (10/12), User Requirements Notation (URN) – Language definition. `http://www.itu.int/rec/T-REC-Z.151/en`
11. jUCMNav – Eclipse plugin for the User Requirements Notation, `http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome`
12. Jensen K., Kristensen L.M. Coloured Petri Nets: Modelling and Validation of Concurrent Systems. // Springer 2009.
13. Stenenko A.A., Nepomniaschy V.A. Model checking approach to verification of coloured Petri nets // Preprint 178.- Institute of Informatics Systems SD RAN.- Novosibirsk.- 2015 (in Russian). `http://www.iis.nsk.su/files/preprints/stenenko_nepomniaschy_178.pdf`
14. Vizovitin N.V. Verification of UCM Models of Distributed Systems with Protected Components Using Coloured Petri Nets. Appendix. `http://bitbucket.org/vizovitin/ucm-verification-examples-2`
15. Vizovitin N.V., Nepomniaschy V.A. UCM-specifications to coloured Petri nets translation algorithms // Preprint 168.- Institute of Informatics Systems SD RAN.- Novosibirsk.- 2012 (in Russian). `http://www.iis.nsk.su/files/preprints/168.pdf`
16. Vizovitin N.V., Nepomniaschy V.A., Stenenko A.A. Verifying UCM Specifications of Distributed Systems Using Colored Petri Nets // Cybernetics and Sys. Anal. 51, 2.- Springer.- 2015.- P. 213-222.