High-level Petri Nets model for XP methodology

Marian Jureczko*, Jan Magott*

*Institute of Computer Engineering, Control and Robotics, Wrocław University of Technology, Wybrzeże Wyspiańskiego 27, 50-370 Wrocław, Poland

Abstract. This paper presents a formal model for software development process under the eXtreme Programming methodology. This model utilizes the High-level Petri Nets and redesigned MAM nets notation to describe workflow and human resources management. Artifacts which are created during the development process and human agents (developers, managers, customers, architects, testers) are represented by Petri nets tokens. This model specifies requirements for each process step: agents and artifacts, and shows the products of execution (newly created artifacts) of each process step. A hierarchy of process steps is presented too.

Keywords: eXtreme Programming, MAM nets, High-level Petri Nets.

1. Introduction

The eXtreme Programming (XP) methodology is usually presented as a set of rules [1], [2], [4], [19]. This approach is very useful when we would like to take some parts of the methodology to improve our software development process. But when we want to create a complex tool which will be supporting the XP methodology in each aspect, we need something much more formal than a set of rules. This paper presents a formal model for the XP methodology.

There are various types of approaches to software development process modeling. Some of them are dedicated to process modeling, like Business Process Modeling Notation (BPMN) [7], some are more universal, like Petri Nets. BPMN is quite new and very interesting approach. BPMN can be translated to an executable language (BPEL, or BPML). Hence, it may be very powerful tool for software development process modeling. There is a possibility to make transformation from model to an application. This application may be used in order to support software development process. Unfortunately BPMN is not perfect. There are difficulties with modeling human resources allocation. Methods of validation and optimization of BPMN's models are still in development phase [7], [20].

Another approach to software development process modeling is an UML profile called Software Process Engineering Metamodel (SPEM) [18]. In SPEM, the following basic elements are defined: Process (it aggregates all elements that define the software development process), WorkProduct (it is everything that is created,
utilized or modified in the Process), WorkDefinition (it is superclass of Activity, Phase and Iteration; it describes the work performed in the process), Activity (it represents a piece of work that is performed by exactly one ProcessRole; Activity is not atomic; it might consist of several atomic elements - steps), Phase (it is a sort of WorkDefinition; it has got precondition, postcondition and a goal called milestone), Iteration (it is a sort of WorkDefinition; the goal of a iteration is a new release of software product), Discipline (it aggregates those Activities that are connected with the same theme), ProcessRole (it is a description of responsibilities and competences of an abstractive existence that performs Activities), ProcessPerformer (it is the performer of WorkDefinitions that are not associated with any ProcessRoles).

Detailed overview of all SPEM elements can be found in [18]. In fact SPEM defines the minimal set of process modelling elements that are necessary to construct software development process model [18]. There are no specific elements or facilities that supports project enactment or analysis. Especially there are no human resource management and the time analysis is very weak [10]. Activities and Steps describes what should be done but neither Activities nor Steps defines time limits. The notion of Phase (the main temporal ordering concept) does precisely some time constraints, but unfortunately Phases are defined on a very high level. There is no possibility to make time analysis that are based on Phases.

Knowledge about UML is absolutely necessary to understand SPEM models because SPEM utilizes almost all UML diagrams. According to [18] only implementation diagrams and component or node diagrams should not be used. Class diagrams are dedicated to represent structure, decomposition, and dependencies of WorkProducts and relations between ProcessPerformer or ProcessRole and WorkProduct. Package diagrams are dedicated to represent Process, ProcessComponents, ProcessPackages, and Disciplines. Use case diagrams show the relationship between ProcessRoles, Phases, WorkDefinitions and Activities. Sequence diagrams shows interaction between SPEM model element instances. Statechart diagrams presents the behavior of SPEM model elements, especially WorkProducts. Activity diagrams can be used to illustrate workflow between Activities with their input and output WorkProducts. Swimlanes separates the competences of different ProcessRoles.

The SPEM specification does not overlap planning, executing and enactment of the software development process (assignment of resources is a part of process enactment). According to [18], the matter of process definition is still under consideration. Additionally, according to [18], there have been plans to apply UML Activity Diagrams (ADs) in this area.

UML ADs have been significantly changed in last years. In versions of UML until UML version 1.5, ADs were based on state-charts. Semantics of UML 2.0 ADs is defined in terms of token flow inspired by Petri nets. However, this semantics is not fully based on Petri nets. In paper [21], UML 2.0 ADs in terms of workflow patterns are analysed. Moreover, there are some works where transformations from ADs to Petri Nets are defined. In [12], a net verification method of discrete systems described by UML 1.4 ADs is proposed. This method is based on conversion of given UML AD into coloured Petri net (CPN). Software tool for automatic conversion has been prepared. Such significant property as lack of deadlocks can be analysed using this method. A theoretical foundation of this analysis is liveness property of CPNs.
Generally, the AD to CPN transformation enables formal verification and validation using tools and techniques available for CPNs. Transformation methodology for UML 2.0 ADs into CPNs is given in paper [9]. CPN are very close to High-level Petri Nets (HLPN) [11]. A transformation of UML 2.0 AD to Object Petri Net is shown in [5].

In paper [6], UML 2.0 based profile for software development process modelling is given. This profile is based on UML 2.0 AD. Hence, it is strongly connected with Petri nets.

There are some Petri net based approaches to software development process modeling. There are various types of Petri nets and some of them are standardised, for example [11]. There are also extensions to Petri nets, which are dedicated to software development process or workflow modelling [14], [16]. The most popular Petri nets based approaches to software development process are FUNSOFT [8] and SPADE [3].

FUNSOFT nets are built on top of Predicate/Transition nets, and they are adapted to the application domain of software development process modeling. FUNSOFT extends the notation of the Predicate/Transition nets, but do not change the semantic power of language. There are some constructions that are dedicated to software development process modeling. FUNSOFT transitions (agencies in FUNSOFT terminology, agencies are dedicated to represent activities) has an input firing behavior, \( F_{IN}(j) \in \{\text{ALL, MULT}\} \), and an output firing behavior, \( F_{OUT}(j) \in \{\text{ALL, SOME, DET, MULT, COMPLEX}\} \). The input firing behavior describes from which input places tokens are removed when a transition is fired. The output firing behavior describes to which output places tokens are added when a transition is fired. FUNSOFT places (channels in FUNSOFT terminology) has got two functions \( S_A \) and \( S_T \), that assigns attributes to places. \( S_A \) assigns an access attribute to each channel, \( S_A \in \{\text{FIFO, LIFO, RANDOM}\} \). \( S_T \) assigns an object type from \( O \) (\( O \) is a set of object types that contains some predefined types and user defined types) to each place. There are also functions that assigns attributes to transitions and arcs. FUNSOFT allows using hierarchy in the model too. Detailed overview of all FUNSOFT features can be found in [8]. Those features simplifies the model of a software development process. FUNSOFT nets analysis facilities cover software development process simulation (with time analysis too), software development process model verification and the evaluation of finished software processes. There was implemented software process management environment, called MELMAC (later reimplemented to a new product - LEU), that support FUNSOFT nets and covers all mentioned facilities. FUNSOFT nets are concentrated on artifact production and flow. There are no facilities dedicated to human resources management. Predicate/Transition nets are enough powerful to create model of software development process with complex human management, but it increases size of the model and makes the model less understandable.

The SPADE project provides a software engineering environment to support Software Process Analysis, Design, and Enactment (SPADE). There is implemented a prototype of the SPADE environment, called SPADE-1. The environment is based on a process modeling language, called SLANG (SPADE Language), which is based on high-level timed Petri net called ER net. ER nets are based on Predicate/Transition nets and Coloured Peti nets. SLANG provides the necessary mechanisms for process modeling without freezing any process-specific policy in the language. It also provides quite interesting modularization construct in an object-oriented style. There
are modules that interacts with the rest of the model through its interfaces when its implementation remains hidden. SLANG allows to describe meta-process and provides features for process modeling and evolution, especially evolution during execution of the process – it supports different policies to apply changes to a process model. SLANG supports the modification of both the process model and the process state during process enactment. SLANG describes interaction with external objects too. SLANG is a powerful, but low-level, process language. The whole software process, which includes the metaprocess, may become quite large and complex. There are no possibility to tolerate light-weight changes to the process rules. Everything must be explicitly defined as a change in the process model. Human interactions and human management are very important in software development processes, unfortunately, SLANG does not provide any facilities dedicated to those areas of processes. Human agents are handled in almost the same way as artifacts.

An agile software development process model is presented in this paper. This model utilizes High-level Petri nets (HLPN) [11] and redesigned MAM nets notation [16] (we propose new internal structure of process step and agent allocation unit). MAM nets are dedicated to modeling software process and organization properties. They give an easy way to create hierarchic constructions and they simplify the model of human agents flow. This model is the first step in developing a Process-centered Software Engineering Environment (PSEE). Process optimization and time analysis may be performed on a HLPN model, so this model may be a base for future works.

Structure of this paper is as follows. In section 2, the definition and graphical notations of our HLPN modification of MAM nets are described. In section 3, our HLPN model for XP methodology is presented. In section 4, conclusions and vision what to do with introduced model are drawn.

2. MAM nets

MAM nets were proposed in [16]. Process example, that has nothing in common with the XP process, is modeled in MAM nets in [17]. They are dedicated to create models of software project development with resources flow. MAM nets are based on Petri Nets. They consist of places, transitions, arcs and components called process steps and agent allocation units.

2.1 MAM nets specification

A MAM net is defined as a tuple (PS, AAU, IP, OP, PI, AI, T, F, P, An, M0), where:

PS is a finite set of process steps. A process step represents an activity in software development processes. Several process steps may be hierarchically abstracted into one large process step.

AAU represents a pool of the currently available human agents. A process step is supposed to send a request to the AAU in order to receive tokens that represent human agents, who are required for the process step's execution. When the execution is finished all allocated tokens come back to the AAU.
IP is a finite set of input-artifact receiving ports of process steps. Each process step has got one input port (ip).

OP is a finite set of output-artifact sending ports of process steps. Each process step has got one output port (op).

PI is a finite set of agent allocation ports of process steps. Each process step has three communication ports: PI = {pq, pc, pl}; pq is a agent-requesting ports, pc is a agent-receiving port, pl is a agent-releasing port.

AI is a finite set of process agent allocation ports of the AAU. AAU also has three communication ports: AI = {aq, as, ar}; aq receives tokens that represent agent requests from pq, as is the port for sending tokens that represents the requested agents to pc, ar receives tokens that represents the released agents from pl.

T is a finite set of transitions.

P is a finite set of places.

F is a finite set of arcs.

An is a finite set of annotations.

M0 is the initial marking.

The MAM nets are high level Petri net formalism that differs from HLPNs and Coloured Petri Nets. The most important part of MAM nets are process steps and agent allocation unit. In this paper those elements are modified in comparison with the original version that was presented in [16] and [17]. In this paper the internal structure of those elements is represented in HLPN formalism [11] (fig. 2, fig. 3).

Types and variables which occur in further showed nets are defined below.

A = {a, c, m, d, t} – type that represents human agents.

D = { ... } – type that represents artifacts which are developed in process steps. Some of those artifacts may be connected with Boolean decisions or with identifier. All available artifacts are described in chapter 3. Subtypes of type D are assigned to places of HLPNs.

S = { ... } – set of story identifiers.

P={...} – set of process steps identifiers.

E = A × P – type that represents human agents assignment to process steps.

s, s1, s2, ..., sm:S

m1, m2, ..., mn: Integer

b, b1, ..., bn: Boolean

a1, a2, ..., an:A

(a,p), (c,p), (m,p), (d,p), (t,p), (ai,p): E

d, d1, d2, ..., dn:D

p, q, q1, q2, ..., qn:P

R = Integer × Integer × Integer × Integer × P – type of resources requests.

(m1,m2,m3,m4,m5,p), (n1,n2,n3,n4,n5,p): R

2.2 MAM nets notation

Process steps and AAU are represented as a rounded rectangles with ports. Ports are represented by circles – places in Petri nets formalism. Each place has got an annotation.
– a set of all available values of tokens which may be stored in this place. Available values of tokens, in places that are used to represent artifacts flow, are subsets of D.

Both process step and AAU are made of HLPN. Process step, which is presented in fig. 2, starts its execution when the d artifact appears in the process step’s input-artifact receiving ports (ip). The t1 transition is fired and a human agent request is generated (this request specifies: m1 – number of architects, m2 – number of customers, m3 – number of managers, m4 – number of developers, m5 – number of testers, p – process step identifier). When all such tokens that represent required human agents are added to the pc port, the t2 transition is fired and a newly created artifact – d2 is added to the op port. All allocated agents are released – tokens are added to the pl port.

The AAU, which is presented in fig. 3, starts its execution when a human agent request appears in the aq port and all such tokens that represent requested human agents are available. Available human agents are represented by tokens in the p1 place. The t1 transition is fired and tokens that represents all requested human agents are added to the as port. Each of those tokens, is connected with an information about process step (p variable) – it determines which process step will get the human agent represented by this token. Tokens that represents human agents, which were released by process steps, are added to the ar port. Then the t2 transition is fired and tokens that represents human agents are added to the buffer with available human agents (p1 place).

Fig. 4 presents examples of the basic process modeling mechanisms. Fig. 4(a) presents two sequential process steps. The output artifact of preceding step is passed to succeeding step. Fig. 4(b) presents connection between process step and AAU. Every process step have to be connected with at least one AAU. In fig. 4(c) two process steps that are hierarchically abstracted into one larger process step are presented.
3. Formal model of XP methodology

Presented model shows the order of generation of new artifacts in extreme programming and human resources allocation. For each of process steps a group of human agents who are necessary to complete it is specified. Previously defined type $A$ of human agents contains the following elements:

$a$ – architect, designer. Architects take a part in preparing new story (story is a sequence of actions of the system and of user of the system; the implementation of a story creates a new feature in the system), they create UML models (if any are needed), and they take care about the architecture of currently developed system [4].

$c$ – customer. Customer is the person who orders and buys the system and, what is very important, only he knows what the system should be able to do.

$m$ – manager. Manager is responsible for good communication in team [4]. He makes technical decisions, and together with the customer, he makes business decisions [1]. Several types of managers are defined in XP methodology (according to [4] there are 3 types), but in this model it is made a simplification and all those types of managers are represented by one type.

$d$ – developer. Developers write source code. They usually work in pairs.

$t$ – tester. Testers, together with the customer, prepare acceptance tests. They automatize tests and they test those parts of system that must be tested manually [4].

Artifacts that are created during the software development process are defined below (those artifacts are equal to available values of earlier defined type $D$):

$n_s$ – Not-implemented Story,

$i_s$ – Implemented Story,

$i_{ss}$ – Implemented Stories,

$v_a$ – Version of an Application,

$p_s$ – Project of a Story (story without acceptance tests and without estimation of
implementation time),

\( n_s e \) – Not-implemented Story with time Estimation,
\( n_s t \) – Not-implemented Story with acceptance Tests,
\( s_i p \) – Story with Implementation Project,
\( s_d t \) – Story Divided to Tasks,
\( n_i s \) – Not tested but Implemented Story,
\( s_b t \) – Story with Bug in Tests,
\( s_f t \) – Story with Fixed up Tests.

The assumptions of this model (those assumptions have been made to simplify model):

- There are no partial order between stories (relations like predecessor or successor)
- The artifacts management is not considered (only the artifacts flow is considered)
- The minimal number of human agents that are necessary in process step is known and used in the model. Sometimes there is a possibility to assign more agents and finish the process step earlier, but we do not consider those possibilities.
- Members inside the groups: architects, customers, managers, developers, testers are not distinguished.

3.1 The highest level of the model.

The highest level of the XP methodology is presented in fig. 5. In input port is an \( v_a \) token. If the project is not based on another system the \( v_a \) represents empty version of application. A project under this methodology is developed in iterations that are represented by the “Iteration of a project” step. A new version of currently developed application (\( v_a \) token) is created as a result of each iteration of the project. Each of those versions may be given to the customer (this possibility is not shown in fig. 5).

End of projects occurs as a result of management or customer decision. This decision is represented by the boolean variable \( b \). Until value of \( b \) is false new iterations of project are performed.

The currently developed version of application may be tested after each iteration. This step is not necessary because the system passed all unit tests and acceptance tests, so we may hope that the application is on quite high quality level. Although if we would like to install the system in customer environment, the testing should be very carefully considered. Many informations about testing version of application can be found in [1].
3.2 Iteration of the project

Fig. 6 illustrates the details of the “Iteration of a project” step from fig. 5. In the XP methodology every project has got iterations. The iteration should not be long, it may last one to three weeks [4]. Iterations are very similar. Sometimes [2] the first iteration is a bit different. There may be put more attention on preparing new stories and planning. In the initial iteration, “The planning game” may apply not only to current iteration but also to the whole project.

In fig. 6, near the input port of the “Iteration of the project”, there is a loop – an arc comes from input port of the “Iteration of the project” to the $t0$ transition and another arc comes back from the $t0$ transition to input port. The $t0$ transition creates copies of version of application. A new copy is created for each instance of the “Preparing a new story” step. In fact there will be exactly as many copies of version of application as is necessary, because after the $t0$ transition, there is a place with bounded capacity (the capacity is bounded to one). The $v_a$ token is removed from input port when a new $v_a$ token is created in the “Refactorization” step (thanks to the $t6$ transition).

The first step of an iteration is the “Preparing a new story” step (detailed model of this step will be discussed later). This step has got an input artifact – version of application. In the first iteration there is an empty version. In the further iterations there is the version of application that was created in previous iteration. A version of application is the best base for customer to invent new stories. As a result of the “Preparing a new story” step we have got a new story. Not every story is implemented in the same iteration in which it was created. Some of the stories may be implemented in further iterations. In “The planning game” step, a decision if a story will (or will not) be implemented in current iteration is made. This decision is represented by boolean variable $b1$. In “The planning game” step, a decision if a chosen to implementation story will be the last chosen story in current iteration is made too. This decision is stored in boolean variable called $b2$. In fig. 6 there are two paths above “The planning game” step. Owing to the path with the $pl3$ place, when “The planning game” step starts, its previous execution have to be already finished. It is impossible to have two working instances of “The planning game” step in the same time. The second path adds tokens that represent stories that are not chosen to implementation in currents iteration ($b1=false$) to the $pl4$ place. The $pl4$ place is a buffer for not chosen yet stories.

Next step of iteration is the “Implementation of a story” step. Above this step there are two places ($pl1, pl2$). The $pl1$ place has got outgoing arc with weight – variable $n$. Value of this variable is equal to the number of stories that were chosen to implementation in current iteration of the project. The $pl2$ place contains a token if the story that is the last chosen to implementation in current iteration has been already chosen ($(n,s,b1,b2)$ token with $b2=true$). The “Refactorization” step can not be started until the story that is the last chosen to implementation in current iteration ($(n,s,b1,b2)$ token with $b2=true$) will not be chosen and before exactly $n$ stories will be implemented ($n$ is the number of stories that has been chosen to implementation in current iteration).
Figure 6: Iteration of the project.

$r1 = (0, 1, 1, 0, 0, q1) - \text{customer+manager}$

$r2 = (1, 0, 0, 1, 0, q2) - \text{architect+developer}$
Stories usually create a partially ordered set. But as it was said earlier, we ignore all relations between stories, so in our model the stories may be implemented in any order. All chosen to implementation stories are stored in the same place (input port of the “Implementation of a story” step). If we consider the partially ordering relation, results of [13] can be applied.

The last process step in iteration of a project is refactorization. Refactorization has got two important purposes. It is performed to remove redundance from source code and to make source code as readable as possible. The “Refactorization” process step is responsible just for removing redundance from source code. Making the source code more readable may be performed much earlier. In fact it is performed in the “Implementation of a story” activity (it is performed separately for each story).

3.3 Preparing a new story

Fig. 7 illustrates the details of the “Preparing a new story” step from fig. 6. A well written story should contain estimation of time, which will be necessary to implement this story and acceptance test to validate implementation of this story. The implementation process will be finished when the implementation pass those tests.

Customer is very important in preparing new story. He is the only person who has knowledge about required functionalities. But he is not an expert in computer engineering. He needs help to write stories that will be understandable to programmers and possible to implement. Customer may be supported by a manager. Each story is duplicated. First copy is sent to developers (“Estimating time for implementation process” step) and the second one is sent to testers (“Preparing acceptance tests” step).

Figure 7: Preparing a new story.

r1=(0, 1, 1, 0, 0, q1) - customer+manager
r2=(0, 0, 0, 1, 0, q2) - developer
r3=(0, 1, 0, 1, q3) - customer+tester
Figure 8: Implementation of a story.
\[ r_1 = (1, 0, 0, 0, 0, q_1) \] - architect
\[ r_2 = (1, 0, 0, 1, 0, q_2) \] - architect+developer
\[ r_3 = (0, 0, 0, 2, 0, q_3) \] - 2'developer
\[ r_4 = (0, 0, 0, 1, 1, q_4) \] - tester
\[ r_5 = (0, 0, 0, 0, 1, q_5) \] - tester
After the “Preparing a new story” step, each \( p_s \) token gets an identifier (variable \( s \)). The \( (n_s\_t,s) \) token represents story with acceptance tests and the \( (n_s\_e,s) \) token represents story with time of implementation estimation. When the \( (n_s\_t,s) \) token and the \( (n_s\_e,s) \) token are removed from input places of the \( t_1 \) transition, thanks to the \( s \) identifier, we are sure that those tokens represent the same story.

In the “Estimating time for implementation process” step, developers estimate implementation time. If possible, the estimation should be done by several developers (an essential minimum is presented in fig. 7 – one developer). According to [19] the whole team of developers should work there. Historical data, about previous iterations or projects, may be very helpful in those estimations.

The acceptance tests should be prepared together by customer and tester. Tester is a person who is familiar with automatic testing tool and has a lot of experience in testing and customer knows how should the developed system behave.

### 3.4 Implementation of a story

Fig. 8 illustrates the details of the “Implementation of a story” step from fig. 6. Implementation of a story consists of sequence of activities. Return to one of the previous steps occurs only when an error is detected. The first step of the “Implementation of a story” step is “Designing implementation of a story”. An UML model with a lot of implementation details may be prepared in this step. But according to the XP methodology rules this step should be as short as possible. Architect should design solution only to the most difficult problems, that may be hard to understand by developers.

Story with implementation project \( (s\_i\_p) \) is sent to developers. Developers divide the story into several tasks. Each of the tasks may be implemented separately. When all tasks are implemented the whole story is implemented. “Implementation of the tasks” step is quite complicated process step. It has not been detailed because of lack of space. Implementation of each of the tasks consist of several activities. Each of those activities should be performed by a pair of developers. Pair programming is one of essential rules of XP methodology. First of those activities is preparing unit tests (according to the one of XP rules: “Test first”). When the developers has got unit tests they can start the implementation process. The implementation lasts until each of unit tests will be passed. When all tasks are implemented they are integrated. Developers should integrate the source code as often as possible, not only when they finish implementation of a task.

Implemented story is submitted to acceptance tests. The acceptance tests should be automated. If the automation is not possible, the acceptance tests are performed by testers. The “Acceptance tests” step gives one of several various tokens in its output port: 1) \( i\_s \) token – Implementation of story passed acceptance tests. The “Implementation of story” step may be finished. 2) \( s\_d\_t \) token – There is an error in implementation of story. Process has to go back to the “Implementation of the tasks” step. 3) \( n\_s \) token – There is an error in project of implementation. The story has to go back to the “Designing implementation of a story” step. 4) \( s\_b\_t \) token – There is an error in acceptance test. The story has to go to the “Correcting tests” step and then goes back to the “Acceptance tests” step.
4. Conclusions and future works

In this paper a formal model of the XP methodology has been described. This model utilizes MAM nets. We have suggested a new version of MAM nets that is compatible with HLPN. In this model, some simplification has been done, and some aspects of the software project development process have been skipped (the “Implementation of the tasks” step might be detailed, the artifacts management is not considered, time that is necessary to complete process steps is not given). In order to make more detailed model we need to do further research in the future.

It is planned to apply introduced model in preparing a software tool that can support management of quality assurance in agile software engineering.

Very interesting feature of this model will be time analysis, but to make time analysis we need information about duration of process steps and probability of backward propagation. Backward propagation occurs when acceptance or unit tests fails. Both of those information should be taken from real software projects. If stochastic models of process step duration and probabilities of backward propagation are defined, results from stochastic PERT networks [15] can be applied.

There is some work with scheduling in real projects. Especially if the “no partial order between stories” assumption will be dropped. There are usually partially ordering relations in real projects. A Petri Net model can be helpful in resolving schedule problems [13]. In the future, scheduling may be supported by the described model.

References


