1990

Petri Net-Based Models of Software Engineering Processes

Kraemer, B.; Luqi

Naval Postgraduate School

B. Kraemer and Luqi, "Petri Net-Based Models of Software Engineering Processes", Technical Report NPS 52-90-011, Computer Science Department, Naval Postgraduate School, 1990.

https://hdl.handle.net/10945/65237

T53

NPS52-90-011

NAVAL POSTGRADUATE SCHOOL

Monterey, California

PETRI NET-BASED MODELS OF SOFTWARE ENGINEERING PROCESSES

KRAEMER, BERND and LUQI

August 1989

Prepared For:

National Science Foundation
Washington, DC 20550

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared in conjunction with research funded by the National Science Foundation.

[redacted]

LUQI
Assistant Professor
of Computer Science

Reviewed by:

[redacted]

for ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

[redacted]

KNEALE T. MARSHALL
Dean of Information
and Policy Science

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-90-011	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) 52Lq	7a. NAME OF MONITORING ORGANIZATION National Science Foundation & ONR
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b. ADDRESS (City, State, and ZIP Code) Washington, DC 20550
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundationl 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF CCR-8710737
8c. ADDRESS (City, State, and ZIP Code) Washington, DC 20550	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
PETRI NET-BASED MODELS OF SOFTWARE ENGINEERING PROCESSES (U)

12. PERSONAL AUTHOR(S)
Kraemer, Bernd and Luqi

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Progress	FROM Mar TO Dec 89	August 1989	18

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

We present an extension of the classical Petri model to formally define functional, structural, and dynamic aspects of software engineering processes. In this model Petri nets are augmented with logic specifications that serve to specify the essential static properties of software objects involved in a process and define global constraints to the dynamic behavior of process models. These models have an intuitive, causality-based execution semantics which enables process simulation and formal analysis using tools and techniques that have been developed for a related Petri net-based specification formalism. Structuring mechanisms are provided to support hierarchical decomposition and the systematic combination of separate views of a software engineering process. As an example we model selected aspects of a rapid prototyping process which supports the reuse of archived software components and guides the use of dedicated prototyping tools.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Luqi	22b. TELEPHONE (Include Area Code) (408) 646-2735 22c. OFFICE SYMBOL 52Lq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Petri Net-Based Models of Software Engineering Processes

Bernd Krämer and Luqi
Naval Postgraduate School, Code 52
Monterey, CA 93943
kraemer@nps-cs.arpa

July 11, 1989

Abstract

We present an extension of the classical Petri net model to formally define functional, structural, and dynamic aspects of software engineering processes. In this model Petri nets are augmented with logic specifications that serve to specify the essential static properties of software objects involved in a process and define global constraints to the dynamic behavior of process models. These models have an intuitive, causality-based execution semantics which enables process simulation and formal analysis using tools and techniques that have been developed for a related Petri net-based specification formalism. Structuring mechanisms are provided to support hierarchical decomposition and the systematic combination of separate views of a software engineering process. As an example we model selected aspects of a rapid prototyping process which supports the reuse of archived software components and guides the use of dedicated prototyping tools.

1 Introduction

A criticism of traditional life cycle models has been the subject and motivation of many recent papers arguing for new approaches to software process modeling, e.g., [1, 3, 4, 8]. Rather than paraphrasing their criticism, we restrict ourselves to subsuming evaluation criteria we found in the literature and providing a few supplementary remarks to justify our own approach of a Petri net-based process model (PNP model) and narrow down the range of issues it tackles.

Typical requirements posed to process models are *adequacy* of the model, *readability* and *ease of use*, *hierarchical decomposability*, and amenability to *formal analysis* and *reasoning*. The arguments supporting these requirements are largely obvious, except for the notion of adequacy which is difficult to grasp due to the manifold aspects software engineering processes comprise. They include, for example, management aspects concerning the optimal employment of people and use of material resources, contractual matters, planning and cost issues, communication and synchronization aspects, or methodological concerns aiming at effective development procedures and tool use.

As we can hardly imagine a homogeneous process model capturing all these different aspects in an adequate way, we first discuss in the conceptual framework which the PNP model covers. Basic concepts of the PNP model are described in Section 3. We emphasize a formal approach to specifying the dynamic behavior of software engineering processes and characteristic attributes of software objects and roles of human participants involved. An illustrative example is given in Section 4 where we present two partially overlapping views of a rapid prototyping process that supports evolutionary software development by interactive construction of executable prototypes from reusable software components [12]. In Section 5 we illustrate constructions that allow consistent combinations and stepwise refinements of process model views. In Section 6 the Petri net semantics underlying PNP models is sketched and their potential to allow formal analysis and reasoning, verification, and symbolic simulation is outlined.

2 Behavior-Oriented Software Process Models

Software development is a dynamic and distributed activity in which many cooperating participants may act partially independently of each other to iteratively transform an initial set of requirements into a validated object system. Different participants usually have different and selective knowledge about an evolving software system. The object system is typically characterized by a large set of software components such as requirements definitions, design documentation, specification and program modules, test protocols and the like which coexist at designated development states.

In this context model adequacy means to capture the distributedness and combinatorial nature of information characterizing an object system in its various development states and the distributedness of changes it undergoes. Speaking in technical terms, a process model approach must be able to handle behavioral issues such as *concurrency, synchronization*, and *communication*. It also means to cope with *nondeterminism* occurring in different forms in the course of a development process. For example, resource contention is likely to arise due to the boundedness of resources but often cannot be resolved as a process model is designed; or it might be necessary to specify the range of alternative possibilities to pursue a process execution without being able to provide a deterministic decision procedure becaues it depends on information that cannot be anticipated in sufficient detail.

The dynamic behavior of a process model strongly depends on the structure of software components and specific roles of human participants in that process. Therefore it is crucial to provide abstraction mechanisms that allow the process designer to define *functional* and and *structural* properties of objects and roles at a level of detail that is necessary to understand and control a development process but still admits developers to make design decisions as needs arise.

A suitable abstraction of a program module in the context of version control, for example, might describe its structure as consisting of a name, author, interface, and body attribute. The role of programmers acting as authors of such modules might be sufficiently characterized by access rights determining who is allowed to update which program modules. The behavior of the version control model then would specify at this structural level how and under which conditions these attributes can be changed by processes but would not refer to details of a module body, for instance. These changes include update rights as the team of programmers involved in a project or their roles may change and new modules are constructed as the system evolves.

3 Basic Concepts of the PNP Model

To capture equally well functional, structural, and behavioral aspects of software processes, the PNP model extends the classical Petri net model by object-oriented data abstraction facilities. The latter allow the process designer to introduce different types of software objects and roles of human participants, provide them with distinguishing attributes, and describe functional relations between between them. Petri net concepts serve to adequately specify the rules governing distributed changes to defined attributes and relationships. The combination provides a suitable notion of distributed states and state-dependent actions that can dynamically create new software objects, concurrently change their properties, and delete objects that are no longer needed.

3.1 Static Aspects of Objects

Software objects are treated as typed and uniquely named entities whose structure and properties are expressed in terms of extensible lists of *attributes*. Attributes either are (references to) objects or are data. Data specifications are supported in the PNP model based on typed Horn clause logic similar to the specification approach defined in [10].

New *object types* are introduced by a special form relating a new type name with names and types of attributes which all instances of that object type share. For example, the form

```
object module:  (ext-name,author:name, if:interface, body:impl)
```

defines objects of type `module` to have at least five attributes whose values are of type `name`, `interface`, and `impl` respectively. These attributes might capture those properties of program module relevant for configuration management.

Attribute names like `ext-name`, `author`, `if`, and `body` denote (projection) functions mapping the object type into the corresponding attribute type. Further attributes can be added to an object as needs arise. But they can only be accessed by pattern matching using the following tuple notation for objects:

```
<M,[N,A,I,B,new]>
```

where `new` is such an add-on attribute value.

Similarly to objects, immutable data structures which are composed of a specific list of component data or have a variant type and value can easily be defined using two forms that are inspired by the object-oriented data model introduced in [11]. An example of the first kind is the data structure abstracting from module interfaces as consisting of two lists of facilities that are exported and imported:

```
record interface:  (export,import:[facility])
```

where angle brackets denote a list of items of the type the enclose. An example of the second kind is the following:

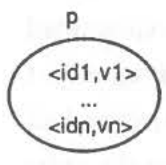
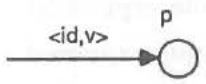
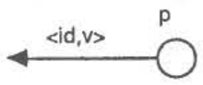
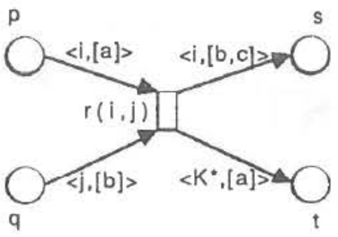
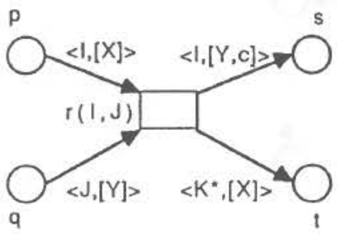
```
variant eval-state:  (new,ckd,vd:unit)
```

It is a trivial variant data structure which just enumerates a finite set of distinct constants used to denote the evaluation status of a software object.

3.2 Dynamic Aspects of Objects

Objects are created dynamically during process execution. Most of the objects created persist as system development proceeds and simply change their attribute values. But there may also be situations in which it is useful to specify that objects are no longer needed and are better discarded. For example, patches to certain program modules can be deleted once a new system version including the dynamically patched changes has been released.

Dynamic object creation, modification, deletion, and changes to mutable relationships among objects and data are captured by *variable predicates* whose extension is changed by occurrences of instances of *actions* which schematically specify similar rules of change

The form	denotes
	the place of all objects <idi,vi> satisfying the variable predicate p in the present development state
	the change element making object <id,v> begin to satisfy p
	the change element making object <id,v> cease to satisfy p
	a rule of change which is symbolized by the term r(i,j) and consists of several change elements including the creation of an object with identity K and attribute list [a] and the deletion of object <j,[b]>; the labeling of arcs expresses the idea that the lifeline of object K starts and that of object j ends with an application of the given rule
 constraining r(I,J) by $X \leq Y$	a scheme of similar rules of change (an action); an instance of an action is obtained by consistently substituting the variables I,J,K,X,Y in the scope of the action by constants such that the formula constraining the action is satisfied according to the specified meaning of functions and predicates it is composed of; note that constants like c express commonalities which all instances share

Notational remarks: Variables are capitalized to distinguish them from function, predicate, and relation names, which are written in lower case.

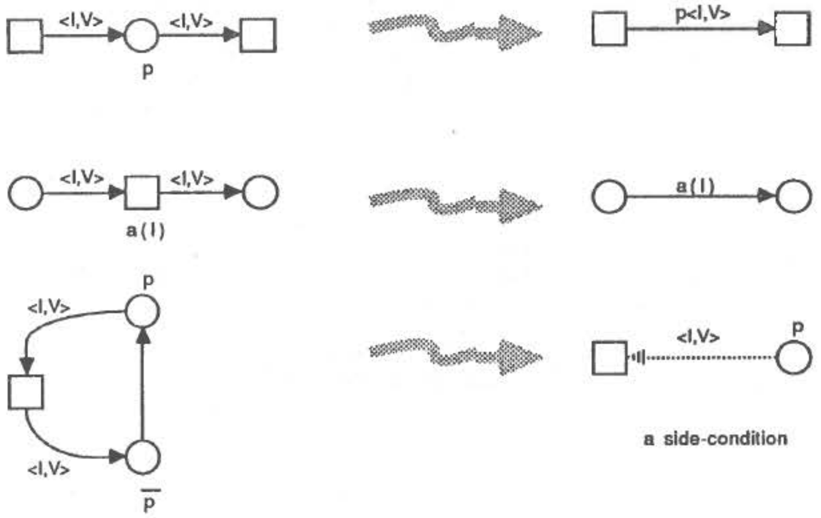
Figure 1: Expressing dynamic aspects of software engineering processes

in distributed processes. As a simple graphical notation for representing these dynamic aspects of software processes, we use Petri nets that are annotated as shown in Fig. 1.

This notation reinterprets basic concepts of high-level Petri nets (see, e.g. [6, 7]) in a specific way to reflect concepts of the application domain. Objects are always represented as pairs. The first component is a unique identity which is implicitly provided as an object is created. It can never be changed and allows one to follow the lifeline of an object and all changes it underwent. The second component is a list of attribute values given in the order determined by the corresponding object definition. Object creation is made explicit by append a * to the variable referring to a new object. Object deletion is simply expressed by letting the lifeline of an object end in an action. A deleted object is no longer accessible in the further course of a software proccess.

3.3 Example

Using this notation and the following abbreviations



a model of a simple version control system providing only one action to release private modules as substitutes for previous versions kept in a public module library can be composed as shown in Fig. 2. The side-condition of this action requires that only those authors may put their private module into the public library who are assigned the right to update library modules of the proper name.

To keep the example simple, it gives only an incomplete view of our simple version control system. This view does not show how new module names are inserted in the library, how private versions are constructed, and how update rights are modified as module names are created or author names change. As we shall see from later sections, this sort of constructing separate and incomplete views of a process model is supported by

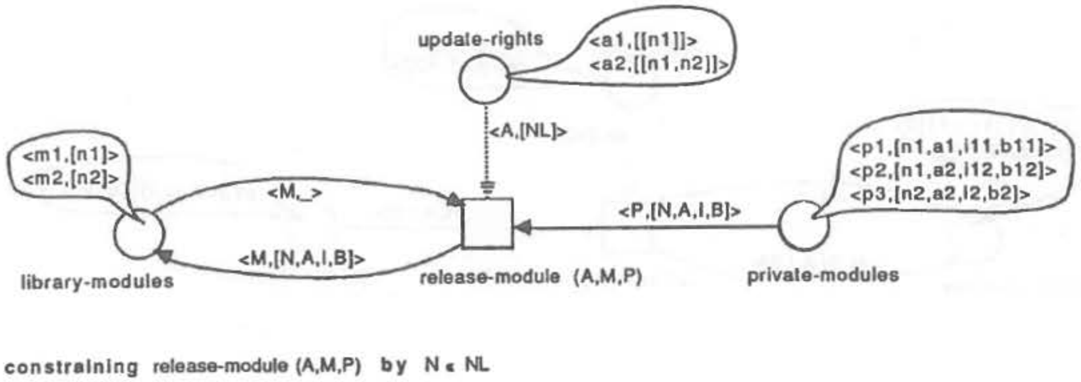


Figure 2: A process model controling the release of private modules as public versions

combination mechanisms that allow one to merge simple views in a consistent way to larger and more complex ones.

3.4 Development States and State Transitions

In a PNP model as shown in Fig. 2 development states are conceived of as distributed entities. Their elements are derived from the variable predicates of a process model and the objects for which those predicates are currently true. In the graphical notation the actual *marking* of a place represents the current extension of a variable predicate. The state given in the example intuitively means that there are currently two public modules named `n1` and `n2` whose contents are still undefined, and we have two authors `a1` and `a2` with `a1` being allowed to update module `n1` and `a2` being allowed to overwrite both `n1` and `n2`; further, there are three private module versions two of which, `p1` and `p2`, are intended to become new public versions named `n1`, whereas `p3` might replace public modules externally known by name `n2`.

Each development state together with the rules of change schematically defined by actions determines the set of possible future states. Transitions between development states are caused by occurrences of instances of actions that are concurrently applicable (see [10] for a formal definition of these concepts). One of the possible future states of our example is shown in Fig. 3. It was caused by occurrences of the changes `release-module(a1,m1,p1)` and `release-module(a2,m2,p3)`. These changes might have happened concurrently according to the given behavior specification. In contrast to this, two other changes that were possible at the initial state, `release-module(a1,m1,p1)` and `release-module(a2,m1,p2)`, mutually exclude each other as they "fight" for the same object named `m1`.

The set of possible states and state transitions is, as usual for Petri nets, defined by the *initial marking*.

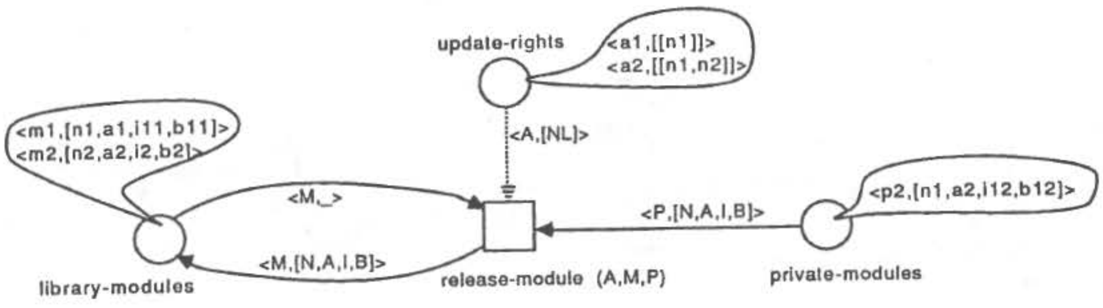


Figure 3: A possible future development state of the process model in Fig 2

4 Formalizing a Rapid Prototyping Process

In this section we develop a PNP model of a rapid prototyping process that supports evolutionary software development by interactive, computer-aided construction of executable prototypes from reusable software components. The model makes previous informal descriptions of this prototyping approach [12] more precise and concrete in that it provides suitable abstractions of software objects and captures causal dependencies and independencies among the actions of the process model. The PNP model also provides a better framework to develop an effective prototyping methodology, improve the functionality of the prototyping support environment [13], and control the proper use of its tools. Two different views of the process model are presented separately in Fig. 4 and 5 to simplify the understanding of the overall process, localize modification, and ease its further elaboration.

First we define some of the object and data types whose instances are involved in the rules of change specified in Fig. 4. These types refer to software concepts presented in [12]:

```
object req-def:  (sysname:name, description:text)
object operator:  (opname:name, spec:specification, state:eval-state)
record specification:  (inputs,outputs,statevars:[name-type])
vpredicate system-requirements(req-def)
           released-psdl-designs(operator)
action     construct(req-def,specification)
           modify(operator,specification)
           refine(operator,req-def,implementation)
           analyze(operator)
```

The process model view presented in Fig. 4 illustrates the principal idea of rapid prototyping to iteratively construct, modify, and refine a series of prototypes, each providing

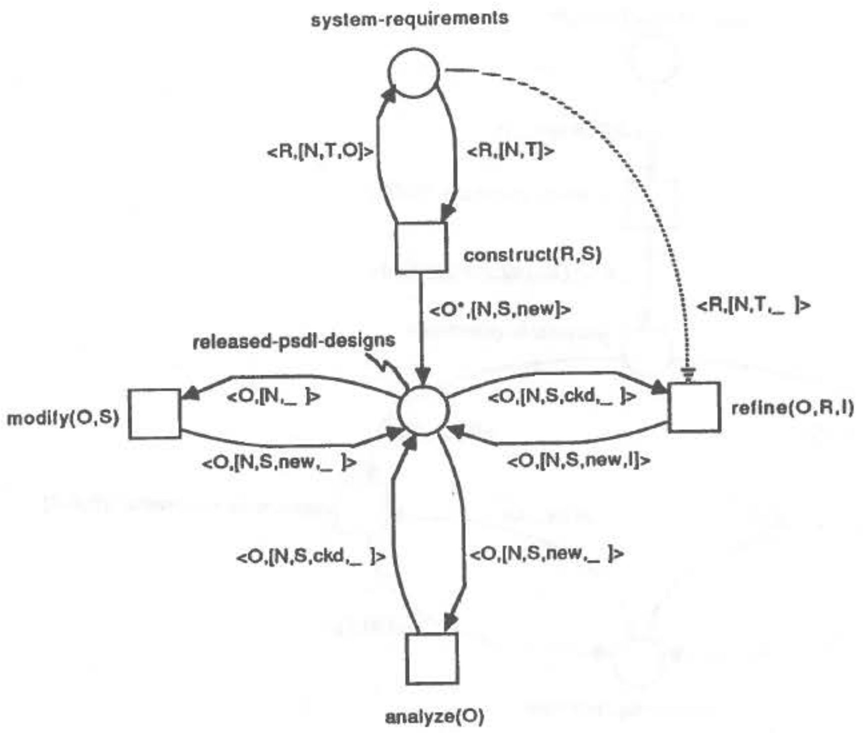


Figure 4: Constructing prototype designs from requirements specifications

the platform for validating and improving previous requirements definitions and design decisions.

At the given simplified abstraction level we do not want to formalize to what extend, for example, the text describing the requirements for a specific system component symbolically named `N` determines the specification of a newly constructed operator realizing these requirements. We just wanted to explicate certain relationships concerning names and references among objects. Looking more carefully at the net labelings, we recognize that certain variables denoting attribute values of objects after a change has occurred are not bound to attribute values existing before that change happened. An example is variable `S` which appears as argument of action `modify` and `construct`. It represents information which cannot be derived from the prehistory of the objects involved but has to be supplied by user of an action. Here the variable represents an arbitrary operator specification which redefines the `spec` attribute of the operator object changed by an instance of these actions. The *information flow* represented by such variables allows us to deal incomplete knowledge in such a way that at least its typical structure and its effect on the the behavior of a process model can be defined.

The process model in Fig. 5 shows another distinguishing aspect of the prototyping process model supporting the PSDL approach [12]. It reveals the role of prototypes to

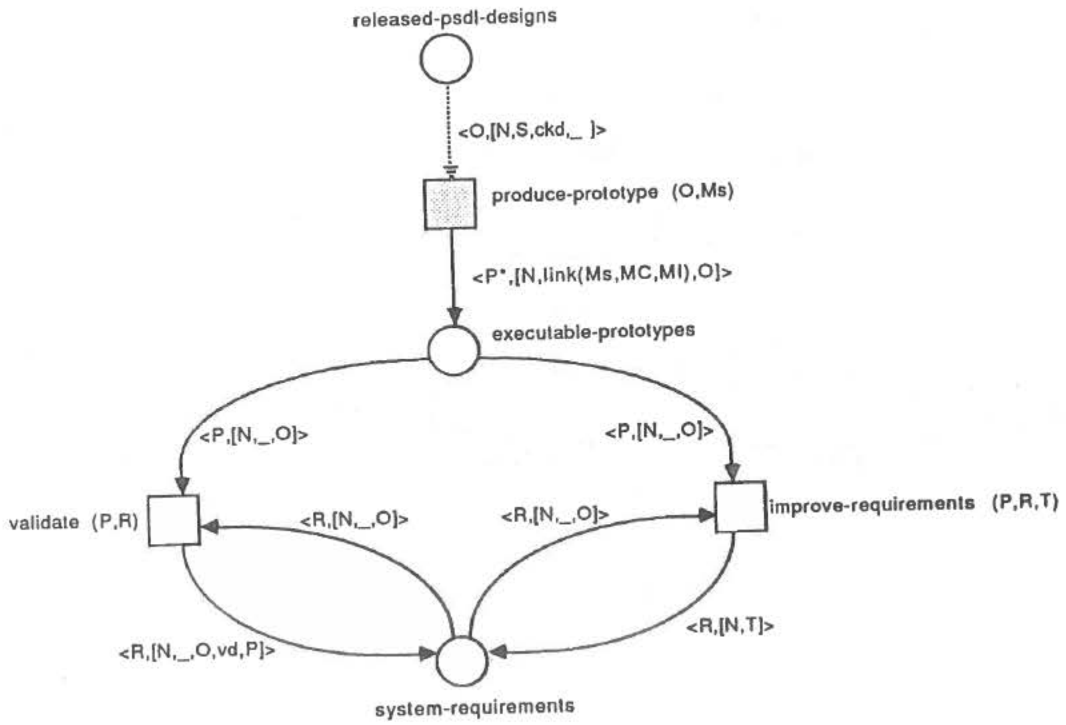


Figure 5: Constructing and evaluating executable prototypes

provide executable models of a proposed system which can be demonstrated to users and customers to validate and improve requirements specifications and design decisions prior producing production code.

5 Horizontal and Vertical Decomposition of PNP Models

One of the primary difficulties in modeling software processes is conceptual complexity. Conceptual complexity can be reduced when the dynamic behavior and the objects of a software process can be composed from independently constructed parts and can systematically be refined. Hierarchical process descriptions are supported by most of the new process models. But horizontal compositions in the sense of combining the parts of a modularized process model are still underdeveloped.

The PNP model approach provides constructions to consistently merge process models representing separate, partially overlapping views of a larger development process. These constructions allow the process designer to

1. synchronize the merged views and connect open information flow lines by identifying actions,

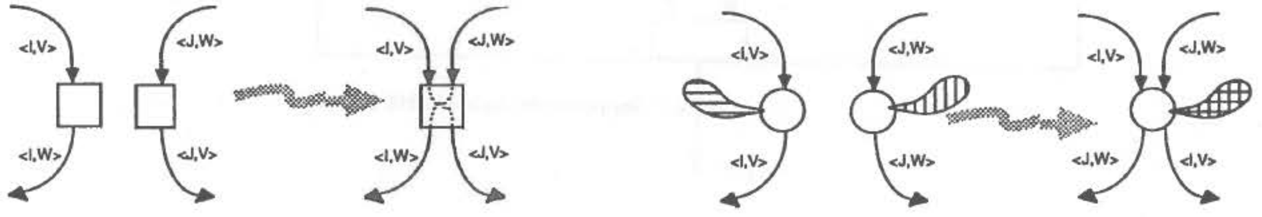
2. combine behavioral alternatives covered in separate views by identifying places and

forming the union of their initial marking, and

3. define new functions operating on objects from different views.

The context conditions to apply these construction and their formal semantics have been developed in the framework of a formal specification language for distributed and concurrent systems [9, 10] and can easily be adapted to PNP models.

Using these constructions the two process models shown in Fig. 4 and 5 can be combined by merging their common places labeled `system-reqirements` and `released-psdl-designs`. The implicit effect of the combination constructions on the behavior of the merged parts is graphically depicted below:



The PNP model also supports stepwise refinements based on substituting actions by subnets whose border only consists of actions, substituting places by subnets whose border contains only places, and abstract implementations of object and data types. Such refinement and implementation concepts have been studied in [14] for the related specification formalism with particular emphasis on defining correctness suitable criteria which provide the basis for verification tools. An example of an action refinement is given in Fig. 6. It shows that action `produce-prototype` can be implemented by two actions working concurrently on appropriate extracts of the object which is input to the abstract action.

6 Semantic Issues and Conclusion

Graphical representations of software concepts have certain advantages in conveying information to human readers but often lack a sufficiently precise semantics to be amenable to formal analysis, verification, and reasoning. One of the strengths of Petri nets is that they provide a simple graphical notation which is easy to comprehend even by non-experienced readers with a strong mathematical background.

The PNP model aims at exploits the comprehensible graphical notation of Petri nets and their precise causality-based execution semantics. It appears relatively easy and straightforward to provide a formal Petri net semantics of the PNP modeling approach by adapting the formal definition of the Petri net based specification formalism we have

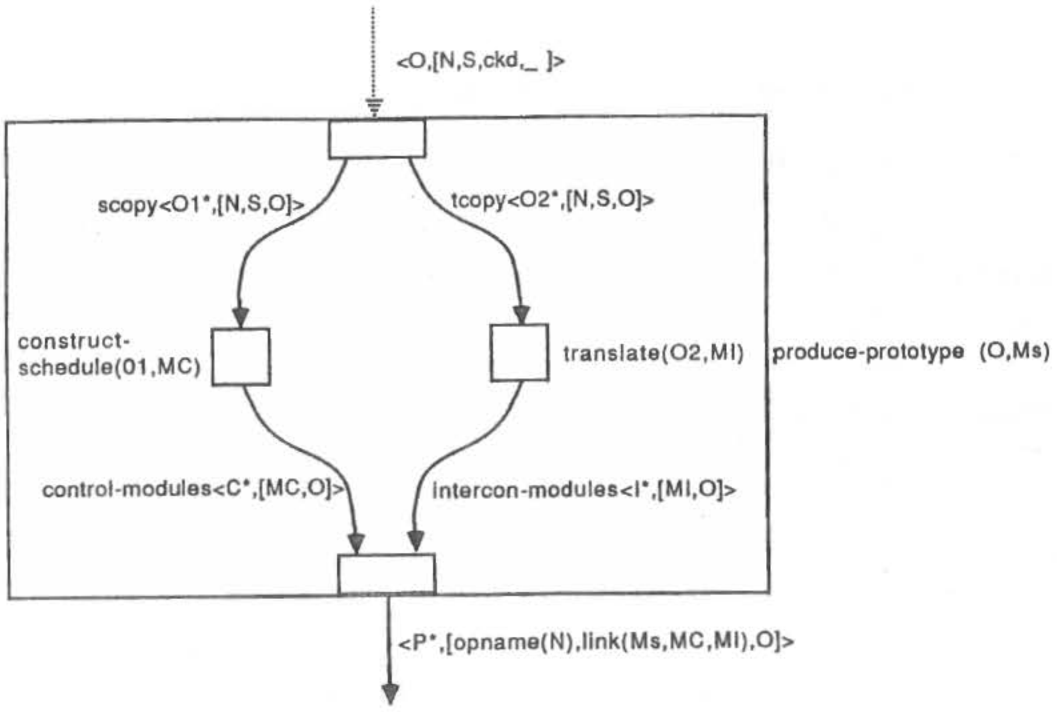

<O,[N,S,ckd,_]>
scopy<O1*,[N,S,O]>
tcopy<O2*,[N,S,O]>
construct-schedule(01,MC)
translate(O2,MI)
produce-prototype (O,Ms)
control-modules<C*,[MC,O]>
intercon-modules<I*,[MI,O]>
<P*,[opname(N),link(Ms,MC,MI),O]>

Figure 6: Refining an action of the process model in Fig. 5

developed earlier [10] to the new concepts introduced here to accommodate specific requirements of software process modeling.

The advantage of such a Petri net semantics would be that theorems, calculi, and validation methods for Petri nets can be reused to support consistency checking, liveness and safeness analysis, verification of the correctness of refinements [14], and invariant analysis techniques [5] for PNP models, too. Liveness and safeness analysis techniques, for example, would help to ensure the continuity of development activities and to prevent overload situations prior to executing a given process model. Or algorithms that generate and analyze the reachability structure of Petri nets might be adapted to support reasoning about behavioral possibilities and inherent facts of a process model.

The Petri net semantics also provides the basis for process model animation using a symbolic simulator for high-level Petri net specifications [2] which redistributes and rewrites objects according to the specified rules of change. Symbolic executions might help to get insight into the behavior of a the specified process and investigate the effects of alternative procedures prior to the actual execution.

References

[1] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

[2] M. Christ-Neumann, B. Krämer, H. H. Nieters, and H. W. Schmidt. The case environment graspin - user's guide. Technical Report ESPRIT Project GRASPIN 37/1, GMD, 1989.

[3] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[4] A. Finkelstein. "not waving but drowning": Representation schemes for modelling software development. In *Proceedings of the 11th Annual International Conference on Software Engineering*, pages 402–404, Pittsburgh, Pennsylvania, May 1989.

[5] Hartmann Genrich and Kurt Lautenbach. S-invariance in Predicate-Transition nets. In Anastasia Pagnoni and Grzegorz Rozenberg, editors, *Applications and Theory of Petri Nets*, number 66 in Informatik-Fachberichte, pages 98–111, Berlin, Heidelberg, New York, Tokyo, 1983. Springer-Verlag.

[6] Hartmann J. Genrich. Net theory and application. In H.-J. Kugler, editor, *INFORMATION PROCESSING 86*, pages 823–831, Amsterdam, The Netherlands, 1986. IFIP, Elsevier Science Publishers B.V.

[7] Hartmann J. Genrich. Predicate/Transition nets. In W. Brauer, W. Reisig, and Rozenberg G., editors, *Petri Nets: Central Models and Their Properties*, number 254 in Lecture Notes in Computer Science, pages 207–247, Berlin, Heidelberg, New York, 1987. Springer-Verlag.

[8] W.S. Humphrey and M.I. Kellner. Software process modeling: Principles of entity process models. In *Proceedings of the 11th Annual International Conference on Software Engineering*, pages 331–342, Pittsburgh, Pennsylvania, May 1989.

[9] Bernd Krämer. *SEGRAS* – a formal language combining Petri nets and Abstract Data Types for specifying distributed systems. In *Proceedings of the 9th Annual International Conference on Software Engineering*, pages 116–125, Monterey, California, March 1987.

[10] Bernd Krämer. *Concepts, Syntax and Semantics of SEGRAS - A Specification Language for Distributed Systems.* GMD-Bericht. Oldenbourg Verlag, München, Wien, 1989.

[11] Bernd Krämer and Heinz-Wilhelm Schmidt. Object-oriented development of integrated programming environments with ASDL. *IEEE Software*, January 1989.

[12] Luqi. Software evolution via rapid prototyping. *IEEE Computer*, pages 13–25, May 1989.

[13] Luqi and Y. Lee. Interactive control of prototyping processes. In *Proc. COMPSAC 89*, Orlando, September 1989.

[14] Heinz-Wilhelm Schmidt. *Specification and Correct Implementation of Non-Sequential Systems Combining Abstract Data Types and Petri Nets.* GMD-Bericht. Oldenbourg Verlag, München, Wien, 1989.

DISTRIBUTION LIST

(1)	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
(2)	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
(3)	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
(4)	Director of Research Administration Attn: Prof. Howard Code 012 Naval Postgraduate School Monterey, CA 93943	1
(5)	Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
(6)	Chief of Naval Research 800 N. Quincy Street Arlington, Virginia 22217	1
(7)	National Science Foundation Division of Computer and Computation Research Attn. Tom Keenan Washington, D.C. 20550	1
(8)	Naval Postgraduate School Code 52Lq Computer Science Department Monterey, CA 93943	100