1991-04

Toward Formal Models of Software Engineering Processes

Kraemer, B.; Luqi

Elsevier

B. Kraemer and Luqi, ''Toward Formal Models of Software Engineering Processes", Journal of Systems and Software, April 1991, Vol. 15, No. 1, pp. 63-74.
https://hdl.handle.net/10945/65708

Toward Formal Models of Software Engineering Processes†

Bernd Krämer‡ and Luqi
Computer Science Department, Naval Postgraduate School, Monterey, California

In this paper, a Petri net-based formal specification method for distributed systems is applied to the domain of software process modeling. We introduce domain-specific concepts stressing the distributed and dynamic nature of software processes. Development states are viewed as distributed entities. Development activities are characterized by their effects on software objects, pertinent information exchange with human or technical participants involved, and local changes to development states. These dynamic aspects of software processes are represented as labeled Petri nets. Structuring mechanisms are sketched which support hierarchical decomposition and systematic combinations of separate views of a software engineering process.

1. INTRODUCTION

Criticism of traditional life cycle models has been the subject of many recent papers arguing for new approaches to software process modeling [1, 5, 7]. Rather than paraphrasing their criticism, we cite evaluation criteria reported in the literature to justify our own approach of a Petri net-based process model (PNP model).

Typical requirements for process models are *adequacy* of the model, *readability* and *ease of use*, *hierarchical decomposability*, and amenability to *formal analysis* and *reasoning*. These requirements are obvious, except for the notion of adequacy which is difficult to grasp due to the numerous aspects of software engineering processes. These include management aspects concerning the optimal employment of people and use of material resources, contractual matters, planning and cost issues, communication and synchronization aspects and 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 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, characteristic attributes of software objects, and tasks of human participants involved. We claim that formalism in software process modeling contributes to consistent and precise understanding of software processes, enables automated support to enhance the reliability and reusability of process models, and opens ways to automate well-understood parts of software processes. Our approach does not address human factors and social processes which might contribute to the software process dynamics [3]. An illustrative example is given in Section 4 where we present two 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 the potential of the model to formal analysis and reasoning, verification, and symbolic simulation is outlined. Section 7 presents our conclusions and discusses the planned extension of the PNP model by a methodology for planning and scheduling the evolution of a software system.

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

Address correspondence to Luqi, Computer Science Dept., Naval Postgraduate School, Monterey, CA 93943.

†This research was supported in part by the National Science Foundation under grant number CCR-8710737 and by the Naval Ocean Systems Center under contract number N6600189WRB0355.

‡This work was conducted while the author was on leave from GMD, D-5205 Sankt Augustin 1, West Germany. Previous work of this author was partially sponsored by the Commission of the European Communities under the ESPRIT project 125 GRASPIN.

655 Avenue of the Americas, New York, NY 10010

0164-1212/91/$3.50

operational system. Different participants usually have different and selective knowledge about an evolving software system. The software system is typically characterized by a large set of software objects such as requirements definitions, design documentation, specification and program modules, test protocols and the like which coexist at designated development states. Semantic relationships between these objects influence the process dynamics and are themselves subject to dynamic changes.

In this context an adequate model should capture the distributed nature of information characterizing an object system in its various development states and the distributedness of the changes it undergoes. Once *concurrency* has been identified as a central issue, a process model is needed that is able to describe the *synchronization* and *communication* necessary to coordinate concurrent activities. Incomplete knowledge introduces the need to cope with *nondeterminism*, which may occur 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 because 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 information provided by tools or human participants as a process is executed. Therefore it is crucial to provide abstraction mechanisms that allow the process designer to define *functional* and *structural* properties of objects and of the information provided at execution time. These aspects of software processes must be specified at a level of detail that is sufficient to understand and control a development process but still gives developers a range of possibilities 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 the attributes author, interface, body, and creation date. The functions performed by the 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 abstraction 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 tasks may change and new modules are constructed as the system evolves. This information cannot be fully determined prior to process execution, nor can it be derived from the process history at a given development state. What we might want to know, however, is the type of information to be supplied and what constraints it might have to satisfy.

3. BASIC CONCEPTS OF PNP MODELS

We describe functional, behavioral, and data aspects of software processes by adapting the SEGRAS specification method [9] to the conceptual framework discussed previously. The PNP model extends this method by introducing an object-oriented data abstraction facility and a restricted form of behavior specifications to enable domain-specific consistency, completeness, and plausibility checks. The difference between data and objects is that data values are immutable, while objects can have states which are subject to change.

The object abstraction facility allows the process designer to introduce different types of software objects, provide them with distinguishing attributes, and describe functional relations between them. Labeled Petri nets are used to specify the rules governing dynamic changes to object attributes and relationships. The combination provides a suitable notion of distributed development *states* and state-dependent and state-changing *actions* that can dynamically create new software objects, concurrently change their attributes, and delete objects that are no longer needed.

3.1 Object and Data Definition

Software objects are treated as typed and uniquely named entities whose structure and properties are expressed in terms of extensible lists of *attributes*. Attributes are either (references to) objects or data. *Object types* are defined through 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: (author:name, spec: interface,
                body:implementation)
```

defines objects of type `module` to have at least three attributes whose values are of type `name`, `interface`, and `implementation`, respectively. These attributes might capture the properties of program modules relevant for configuration management. Further attributes can be added to an object during process execution as defined by the process model.

Objects are represented by pairs. The first component denotes a unique, system-provided object identifier and the second is a list of terms defining an object's

actual attribute values. The elements in the attribute list are given in the order of attribute declarations. An example is the following:

```
<module#263,
 [tom, my-spec, my-body, unchecked]>
```

where term `tom` denotes the author, `my-spec` the actual interface specification, and `my-body` the actual implementation body of module `M`. Attribute `unchecked` is an additional attribute value meant to express the verification state of the module object. Attributes can be accessed by pattern matching. For example, the expression

```
<M, [A, S, B, unchecked]>
```

matches all objects of type `module` that are still unchecked. To match a whole sublist of actual attribute values, we provide the asterisk character `*`. This allows us to use expressions like the following

```
<M, [don, *]>
```

which matches all module objects authored by `don`. Declared attributes can also be accessed by using attribute names like `author`, `spec`, and `body` as projection functions mapping the object type into the corresponding attribute type. For example, if `my-mod` denotes the above `module` object, the term `author(my-mod)` is equal to `tom`.

Similarly to objects, immutable data structures which are composed of a specific list of data components 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 Kramer and Schmidt [10]. An example of the first kind is the data structure representing simplified module interfaces as two lists of facilities that are exported and imported:

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

where square brackets denote a list of items of the type they enclose. A unique constructor function with name `mk-interface`, arity types `[facility]` and `[facility]`, and result type `interface` is automatically derived from the type name. An example for a variant type definition is the following:

```
variant verification-state: (unchecked,
        statically-checked, verified: unit)
```

This variant data structure enumerates a finite set of distinct constants used to record the evaluation status of a software object of type `unit`. The injection functions `unchecked`, `statically-ckd`, and `verified:unit` are taken as the set constructor functions for variant type `verification-state`. Boolean-valued inspection functions named `is-unchecked`, etc., and partially defined selector functions named as `-unchecked`, etc., are also provided automatically. User-defined abstract types can also be specified using the algebraic sub-language of SEGRAS [9].

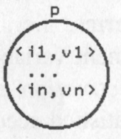
3.2 Process Model Behavior

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

All dynamic aspects of objects are captured in a graphical `behavior specification` given in terms of marked high-level Petri nets. A Petri net can be viewed as directed bipartite graph composed of two kinds of nodes which are called *S-elements* and *T-elements* (see Schmidt [16] for a formal definition). S-elements represent local states and T-elements represent transitions. In the PNP model, all S-elements are labeled with names of unary predicates that are defined on user-defined object types. T-elements are labeled with action names. Arcs, which represent a causal flow relation between S- and T-elements, and vice versa, are labeled with sets of pairs of the form `<Id, Attr-list>` where each pair denotes an instance of a defined object type, `Id` is a unique object identifier and `Attr-list` is a list of terms denoting attribute values of this object. The attribute values are given in the order determined by the corresponding object definition and the causal order of supplementing add-on attributes. The object identity is implicitly provided as an object is created and can never be changed. It allows one to trace the history of changes an object underwent.

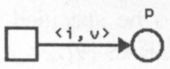
The *marking* of a PNP net is given as a distribution of sets of objects over the S-elements of the net. Markings represent distributed development states.

A labeled S-element such as

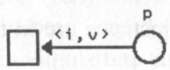


denotes a component of a distributed development state and can be viewed as the definition of a state dependent predicate p which is true for precisely the set of objects associated with the S-element by a marking. The values of state-dependent predicates can be changed by processes.

Labeled arcs express two types of atomic changes:

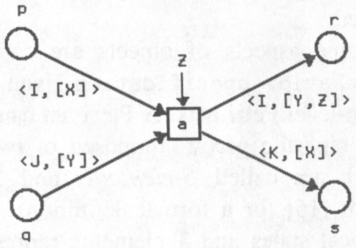


denotes the atomic change that object <id, v> begins to satisfy the predicate p, while the form



denotes the atomic change that object <id, v> ceases to satisfy p.

The form



defines the state change due to an action named a. It describes a scheme of similar rules of changes expressed by using variables in arc labels. We use uppercase symbols to denote variables, while function, attribute, predicate, and action names begin with a lower case letter. We call the set of variables occurring on the arcs adjacent to the action a the *environment variables of* a. The variables in name positions of object tuples schematically describe which objects are affected by a (here the objects named I, J, and K). Attribute values on external arcs such as Z specify *actual arguments* that must be provided when action a is about to happen. All attribute values of objects on input arcs are available as outputs of an occurrence of a. Together with the actual arguments of a they form the set of known attribute values from which the attribute values of objects on output arcs can be constructed.

An instance of action is obtained by consistently substituting ground terms, i.e., terms containing no variables, for environment variables (I, J, K, X, Y, and Z) such that the *activation constraint* "**constrain** a **by** $X \leq Y$ **end**" is satisfied according to the specified meaning of functions and predicates appearing in the constraint. Note that activation constraints generally need not be given.

The notation of objects allows us to determine for each action whether an object is deleted, created, or survives the changes it specifies. Deletion occurs when an object on one of the incoming arcs does not occur on any outgoing arc, while creation occurs when an object on one of the output arcs does not occur on any incoming arc of the action. To make object creation and deletion explicit and to provide checking redundancy, we append a plus (+) or a minus sign (−) to the identifier of an object to be created or deleted, respectively.

Objects are non-distributable entities but knowledge about objects can be distributed in the form of object names occurring as attribute values of other objects. This may even lead to the situations where names of objects that are already deleted are still known. Access to an object's attributes, however, is only possible through participation in the same action occurrences.

3.3 An Example

To illustrate our concept of process model behavior, Figure 1 depicts the behavior of a very simple version-control system. This system provides two actions only. Action establish-new-module serves to release initial versions of modules and to assign the right to update a new module to a specific programmer in the development team. The initial versions have just an interface specification but no implementation body. Action update-module allows authorized programmers to update public versions of modules by implementation bodies of their private versions. In the given development state, we have two public and three private modules, and three authors a1, a2, a3, who are allowed to update module m1, m2, and m1, respectively. As we shall see in the following subsection, the specified behavior allows concurrent updates, provided that they involve different modules. Similarly, new modules with update rights for different authors can be established concurrently.

To keep the example simple we give only a partial view of the version control system. This view, for example, does not show how private versions are constructed and how update rights are modified independently of establishing new modules. As we shall see from later sections, this sort of constructing separate and incomplete views of a process model is supported by composition mechanisms that allow one to merge simple views in a consistent way to form larger and more complex ones. Further we assume the object- and data-type specifications given in Section 3.1 are included in the definition part.

In this example, we further use a special notation for *mutable side-conditions* of actions by means of dashed arcs (see Figure 1 and Figure 4) and for activation constraints using the keywords **constrain** and **end**. Side-conditions are just an abbreviation for inputs of actions that are returned to the S-element when the

action is completed, possibly with new attribute values. Here the side-condition expresses the requirement that only authorized authors may perform an update action. The effect of this side-condition can be changed dynamically by an occurrence of action `establish-new-module`. The current distributed state of the version control system is represented by markings of the places with object instances whose attribute values are given by ground terms. Markings are visually related to specific state elements by dotted lines. The activation constraint associated with the net constrains the set of admissible instances of action `update-module` to those authors `A` whose module update list `ML` contains the identifier of the module to be updated (here `M`). In our example, the identifier of a new module is appended to the module update list of a specific author as the module is established.

3.4 Process Model Dynamics

In a PNP model as shown in Figure 1, development states are conceived of as distributed entities. Their elements are derived from the state-dependent predicates of a process model and the objects for which those predicates are currently satisfied.

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 activated. Informally speaking, an instance of an action a is activated in a given development state if the instance satisfies the activation constraint of a (if any), if all objects labeling incoming arcs of that instance are in the marking of the adjacent S-element, and if all objects labeling outgoing arcs satisfy the predicate labeling the adjacent S-element. The state change affected by an activated instance of an action is determined as follows: from each incoming arc, the object denoted by its labeling is removed from the marking of the adjacent S-element and for each outgoing arc the object in its labeling is added to the marking of the adjacent S-element. All of the changes to the markings associated with an instance of an action are made as a single atomic action. In Kramer [9], a formal definition of these concepts is given for the high-level Petri net language used in SEGRAS. Similar definitions apply in a formal framework for PNP models.

In Figure 1, three instances of action `update-module` resulting from the substitutions {a1 / A, m1 /

```
record author: (authorized: [module]).
actions establish (module,interface,author),
        update (author,module,module).
predicates may-update(author),
           private(module), public (module).
```

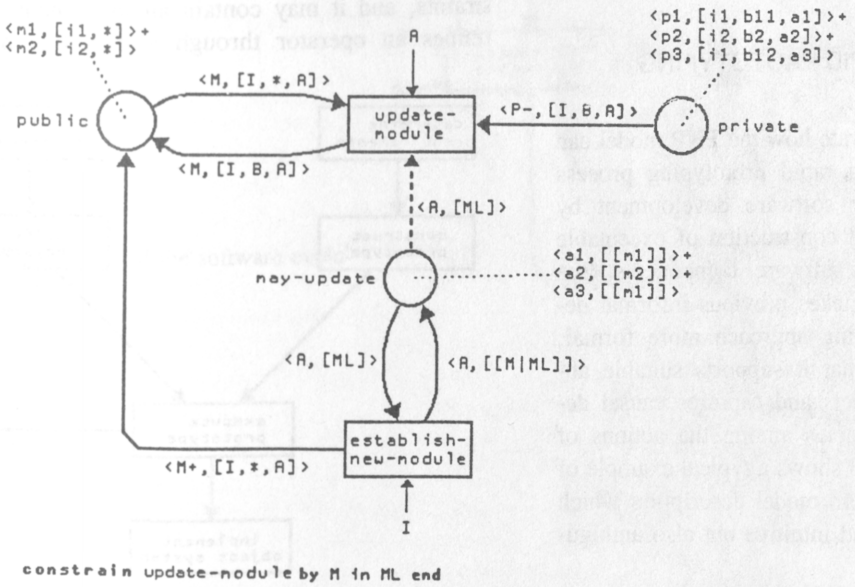


Figure 1. A simple process model controlling the release and update of public versions.

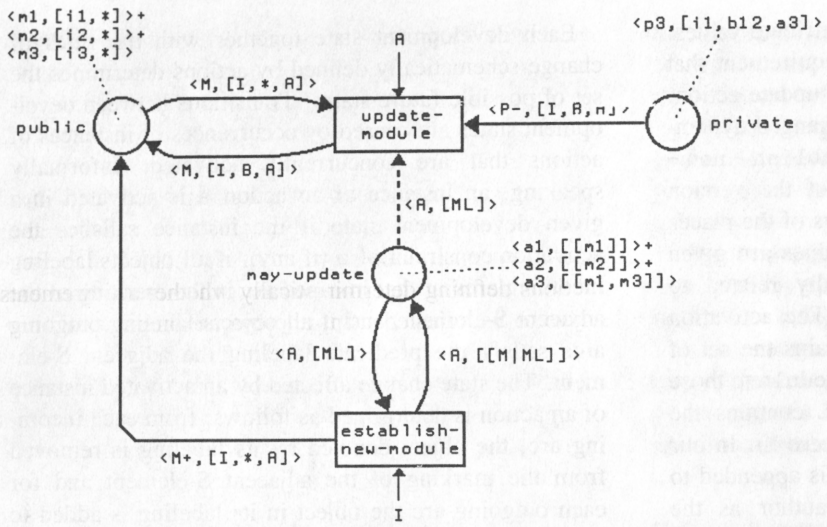


Figure 2. A possible future development state of the process model in Figure 1.

M, p1 / P}, {a2 / A, m2 / M, p2 / P}, and {a3 / A, m1 / M, p3 / P} and infinitely many instances of action establish - new - module are activated.

One of the possible future states of our example is shown in Figure 2. It was caused by occurrences of the first two instances of action update - module and the instance of establish - new - module resulting from the substitution {m3 / M, i3 / I}. These changes might have happened concurrently according to the given behavior specification. In contrast to this, two other changes that were possible at the given state, namely those affected by the first and last instance of update - module mentioned above, mutually exclude each other because they "fight" for the same object named m1.

4. FORMALIZING A RAPID PROTOTYPING PROCESS

In this section we demonstrate how the PNP model can be applied to describing a rapid prototyping process that supports evolutionary software development by interactive, computer-aided construction of executable prototypes from reusable software components (see Luqi [12]). The exercise makes previous informal descriptions of this prototyping approach more formal, concrete, and precise in that it supports suitable abstractions of software objects and captures causal dependencies and independencies among the actions of the process model. Figure 3 shows a typical example of this informal kind of process model description which can be appealing simple and intuitive but also ambiguous and imprecise.

We claim that the PNP model approach provides a basis for increasing the effectiveness of the prototyping methodology by better understanding and insight, improving the functionality of the prototyping support environment [13], providing better user guidance, and controlling the application of its tools. Before we develop a PNP model of Figure 3, we define some object and data types.

The type definitions in Figure 5 refer to software concepts presented in Luqi [12]. A major component of this prototyping approach is the language PSDL used to describe prototype designs as networks of operators connected by data streams. These data-flow networks are augmented with timing and control constraints. Operator definitions consist of a name, a specification of input/output data, internal state variables, and constraints, and it may contain an implementation which refines an operator through a data-flow network of

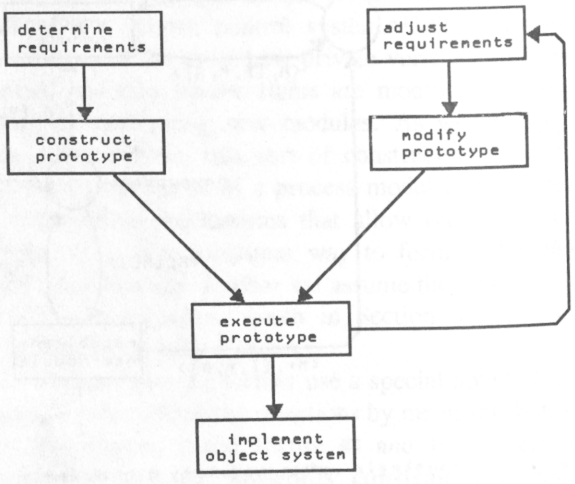


Figure 3. A process model for software evolution through prototyping.

J. SYSTEMS SOFTWARE
1991; 15: 63-74

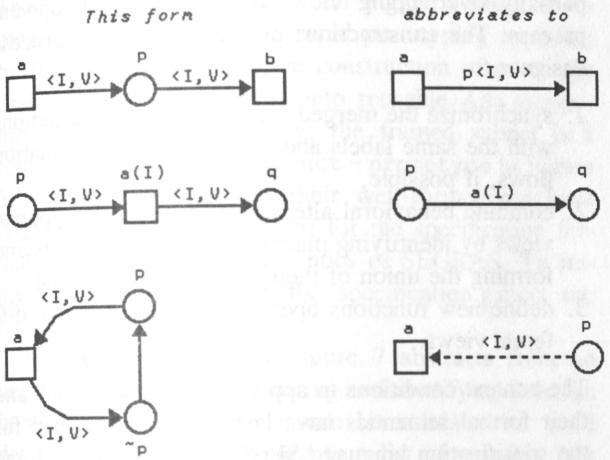


Figure 4. Abbreviations used in PNP models.

other operators. Our data specification below reflects this structure of PSDL descriptions in a simplified form and we assume some types like `text` and `name` to be defined elsewhere. To simplify the graphical presentation of PNP nets, we use the abbreviations depicted in Figure 4.

Figure 5 shows a PNP model corresponding to Figure 3. This model reveals the nondeterminism hidden in the informal description in the form of conflicting actions, explicates development states, and visualizes the information flow and feedback between different actions in the form of object schemes. It also shows relations between the various types of documents and how they are updated by development activities. In the given process model, actions `implement-object-system` and `adjust-requirements` are in conflict with respect to prototype evaluation protocols because there is no activation constraint associated with these actions defining deterministically whether requirements have to be adjusted using the evaluation protocol, previous requirements, and the evaluated prototype as feedback information or whether the intended object system can be implemented. The model just tells us that there is this conflict and that it has to be resolved as the model is refined or when it is executed.

At the given simplified abstraction level, we do not want to formalize to what extent, for example, the text describing the requirements for a specific system component determines the interface specification `S`, the list `Rc` of Ada packages providing reusable code used in prototype construction, or the Ada code `C` of a newly constructed prototype realizing these requirements. We just want to make certain relationships between objects and their attributes explicit. The actual arguments of actions represent information which cannot be derived from the history of the objects involved but has to be supplied by their users. The *information flow* repre-

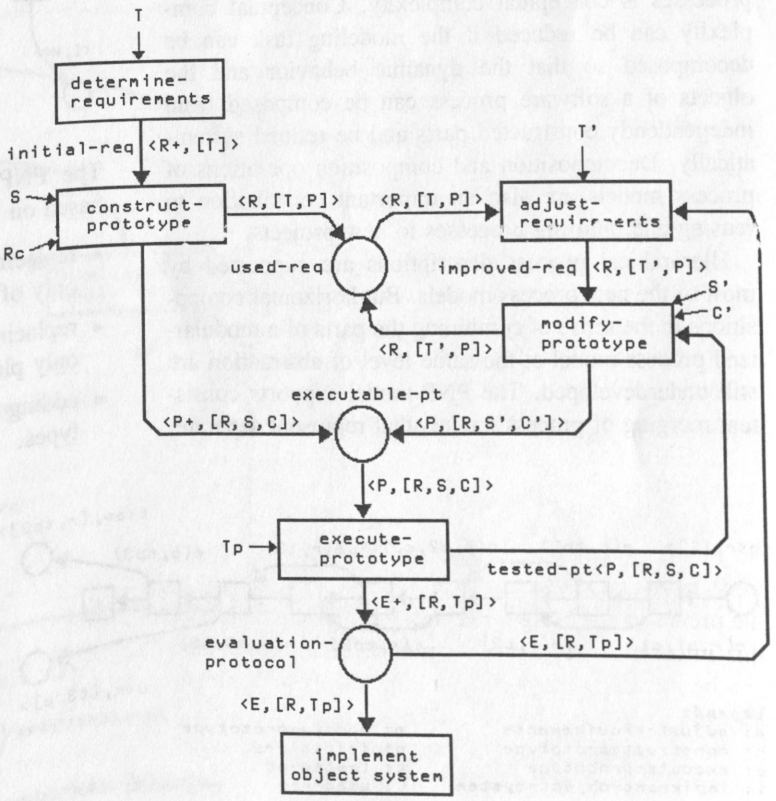


Figure 5. PNP model of the software evolution process model.

sented by such variables allows us to deal with incomplete knowledge in such a way that at least its typical structure and its effect on the behavior of a process model can be fixed. Hence, the type of variable S in action `construct-prototype`, for example, determines the structure of the data denoted by S, while the net specifies the behavioral effect.

A record of the execution history of a process model can again be represented by Petri nets. These nets turn out to be unfoldings of PNP nets. They are acrylic and unbranched in S-elements as each execution of a process model resolves the nondeterministic choices possibly contained in PNP nets. Moreover, their T-elements are labeled with terms denoting the actual instances of actions that were executed and their S-elements with terms denoting the concrete objects involved. Figure 6 shows a record of the execution history of the PNP model in Figure 5. In the recorded execution, the initial requirements had to be adjusted twice and correspondingly the prototypes constructed had to be modified twice before the object system was implemented. The final slice of S-elements depicts the final system state as consisting of the requirements definitions that were successfully evaluated and the final prototype.

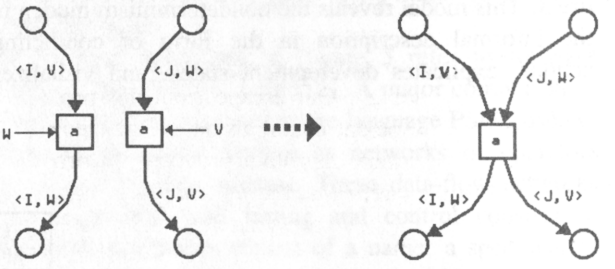
5. HORIZONTAL AND VERTICAL DECOMPOSITION OF PNP MODELS

One of the primary difficulties in modeling software processes is conceptual complexity. Conceptual complexity can be reduced if the modeling task can be decomposed so that the dynamic behavior and the objects of a software process can be composed from independently constructed parts and be refined systematically. Decomposition and composition operations of process models are also an important contribution to reusing and tailoring processes to new projects.

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 at the same level of abstraction are still underdeveloped. The PNP model supports consistent merging of process models that represent separate, partially overlapping views of a larger development process. The constructions provided allow the process designer to

1. synchronize the merged views by identifying actions with the same labels and close external information flows, if possible,
2. combine behavioral alternatives covered in separate views by identifying places with the same labels and forming the union of their initial markings, and
3. define new functions operating on objects from different views.

The context conditions to apply these constructions and their formal semantics have been formally defined for the specification language SEGRAS in Kramer [8] and can easily be adapted to PNP models. Intuitively, the combination construction is a gluing of PNP nets in S- or T-elements which requires that the S- or T-elements to be identified have the same label and results in PNP net that forms the union of the markings of common S-elements and of labels of common arcs. The implicit effect of the combination constructions on the behavior of the merged parts is graphically depicted for T-elements below:



The PNP model also supports stepwise *refinements* based on

- replacing actions by subnets whose border consists only of actions,
- replacing places by subnets whose border contains only places, and
- adding abstract implementations of object and data types.

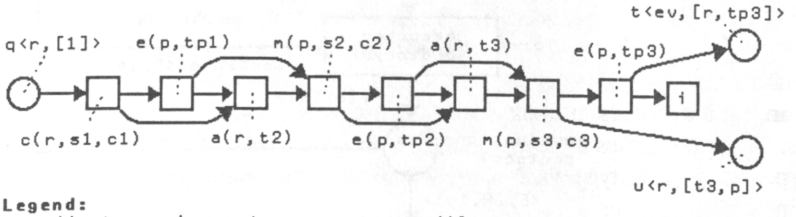


Figure 6. A record of an execution history of the process model shown in Figure 5.

Legend:
a: adjust-requirements
c: construct-prototype
e: execute-prototype
i: implement-object-system
n: modify-prototype
q: initial-req
t: tested-pt
u: used-req

The process model presented in Figure 7 illustrates a more detailed view of the rapid prototyping approach by focusing on the iterative construction of prototype designs and their mapping into reusable Ada components. Technically speaking, the framed subnet is a refinement of action `construct - prototype` in Figure 5. Such refinements and their well-formedness have been studied in Schmidt [15] for the specification formalism underlying the semantics of SEGRAS. To improve the readability of the specification, most arc labels are made invisible.

The process model in Figure 7 abstracts from an iterative process of constructing and modifying a design in terms of PSDL until it can be validated. A valid design can either be mapped to a list of reusable Ada components `Rc` and then turned into an executable prototype named `P` or no reusable component can be identified and the given design has to be rerefined, leading into another iteration circle. The operational part of prototype `P` is Ada code denoted by attribute `C`. However, at the given abstraction level, our process model gives no answer how this code is determined as it is neither (part of) an attribute value of PSDL operator `O` or requirement `R`, nor is it an actual argument of action `produce - prototype`. In the PNP model, this situation indicates a need for further refinement.

The refinement answering the question of how `C` is constructed is given in Figure 8. It shows that the action `produce - prototype`, which appeared in Figure 7, can be implemented by two actions working concurrently on separate copies of a given PSDL design which is input to the abstract action. This refinement reflects a part of the prototyping process which is automated. This process takes a PSDL prototype as input and produces an Ada implementation as output. The Ada implementation results from linking reusable Ada code identified in a software base with generated code implementing the interconnection of operators specified in the prototype. Once a reusable Ada component has been identified, two different tools can be used to generate Ada code from the given PSDL design. A translator uses the data flow links in an operator specification `S` to implement the communication interfaces `c(S)` embedding reusable components `Rc`, which implement operators in `S`. A scheduler attempts to produce a feasible schedule `s(S)` for the execution of time-critical operators. (The possibility that this attempt

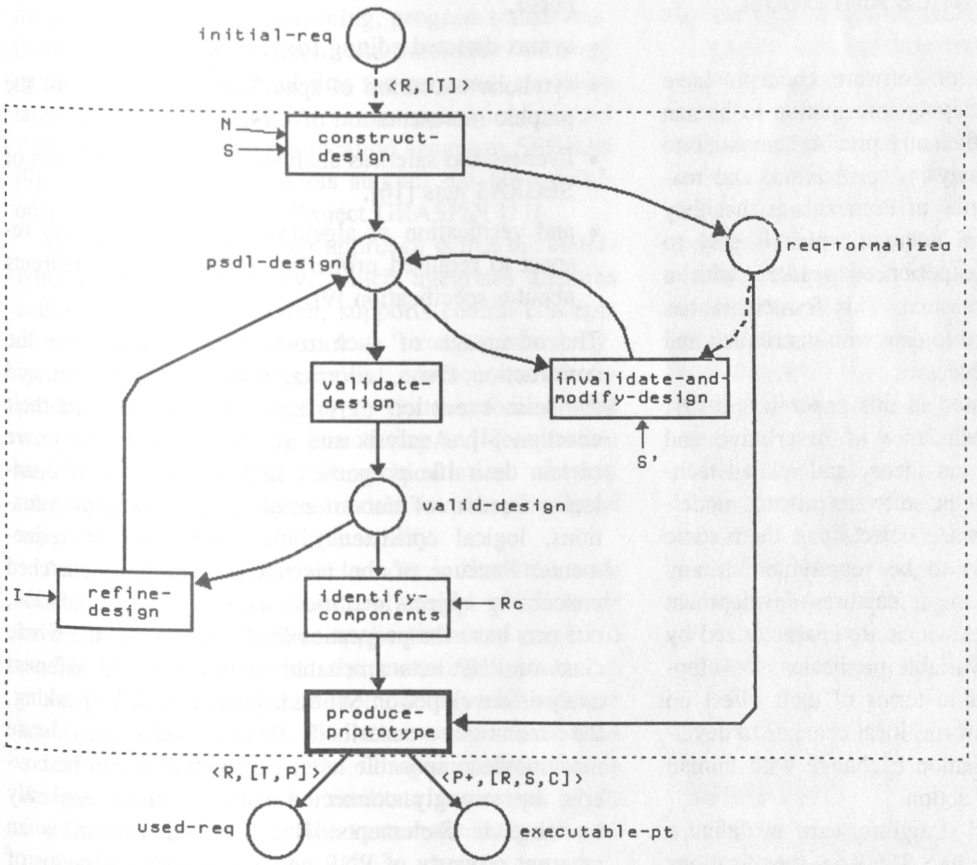


Figure 7. Constructing prototype designs from requirements definitions.

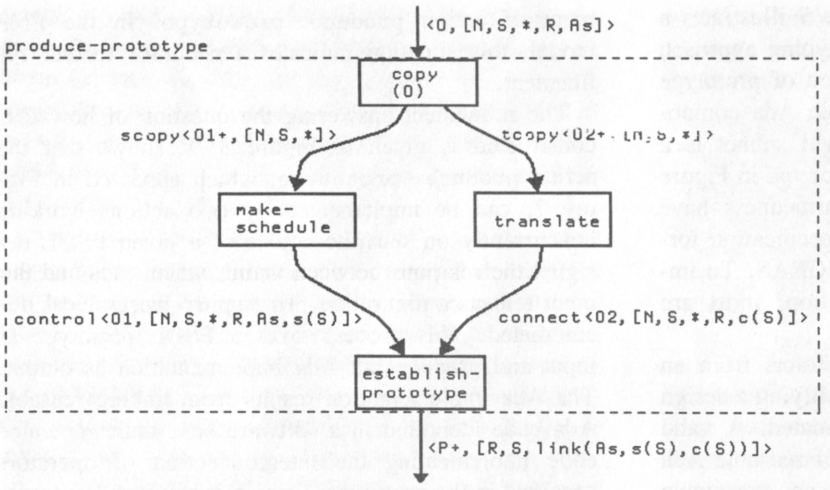


Figure 8. Refining an action of the process model in Figure 7.

may fail is not shown in our example.) The generated Ada packages `c(S)` and `s(S)` then are linked together with the components `Rc` to form the executable Ada code of prototype `P`. The meaning of the functions `s`, `c`, and `link` and the type definitions for Ada components are not given here but are straightforward.

6. UNDERLYING SEMANTICS AND MODEL SUPPORT

Graphical representations of software concepts have certain advantages in conveying information to human readers but often lack 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 inexperienced readers with a strong mathematical background. This framework has been particularly developed to deal with distributed and concurrent systems and processes.

The PNP model presented in this paper was a first attempt to exploit the abundance of descriptive and analytical results of Petri net theory and related techniques and support tools. Our software process modeling approach allows software objects and their static and dynamic relationships to be represented at any desired level of abstraction. It captures development states as distributed entities which are characterized by sets of objects satisfying variable predicates. Development actions are specified in terms of their effect on software objects they transform, local changes to development states, and information exchange with human or technical carriers of an action.

It is relatively easy and straightforward to define a translation of PNP models into SEGRAS specifications for which a formal Petri net semantics already exists [8]. This semantic interpretation of PNP models makes it easy to adapt the construction and analysis tools that are provided by the Graspin environment [11], which supports SEGRAS, to the level of PNP models. Currently, the Graspin environment supports

- interactive structure editing for graphic descriptions, which was used to produce the illustrations in this paper,
- syntax-directed editing for textual specifications,
- symbolic execution of specifications in terms of the graphic representation of nets,
- liveness and safeness analysis for a restricted class of SEGRAS nets [16],
- and verification of algebraic specifications with respect to required properties formulated as theorems about a specification [9].

The advantage of such tools is that they ease the construction, reuse, tailoring, analysis, verification, and symbolic execution of process models prior to their enaction [4]. Analysis and verification serve to prove certain desirable properties such as absence of deadlocks, freedom of starvation, absence of overflow situations, logical consistency, or correctness of refinements. Because of the restrictions we have sketched concerning object identities and object manipulation, our nets have the properties that would make the whole class of PNP nets amenable to liveness and safeness analysis developed in Schmidt [16]. Roughly speaking, the technique described therein relies on state machine-decomposable nets, i.e., nets that can be covered by strongly connected subnets which are only branched in S-elements. But this requirement is an inherent property of PNP nets because the behavior of a process model is composed of the behaviors of sub-

nets which describe the behavior of single objects. 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. Algorithms that generate and analyze the reachability structure of Petri nets can also be adapted to support reasoning about behavioral possibilities and inherent facts of a process model.

Animation of PNP models through symbolic execution allows demonstrations of process models to test their adequacy, provide insight into the dynamic behavior, and thus increase our trust in their design. Symbolic execution can be used to investigate the effects of alternative procedures prior to the application of the process to a real software development project, thus reducing development risks.

7. CONCLUSIONS AND FURTHER RESEARCH

In this paper we have presented a Petri net-based modeling approach for software engineering processes. This approach has several advantages: The software process description language has a formal syntax and semantics which enables precise and unambiguous communication about process models and has the advantage of potential automation including semantic tools for prototyping, formal reasoning, program transformation, performance evaluation, and automatic documentation. In fact, we argued that the proposed modeling language can be considered as an application-oriented variant of the formal specification language SEGRAS [9] for which extensive tools support has been developed within the ESPRIT Project GRASPIN [11].

Another advantage of our approach is that the underlying mathematical theory, which integrates algebras and elementary net systems, supports central concepts of the application domain such as "true" concurrency, synchronization, communication, conflict, causal dependencies, structure, and type. On this formal basis we have been developing structuring and composition capabilities supporting hierarchical and vertical (de)composition of process models for the purpose of separation of concerns and reusability.

An important observation to note is that this work should be seen as an exploratory attempt which needs to be backed up by further applications and empirical studies. This requires that the modeling approach is embedded in a well-defined methodology and software tools support is provided. The latter seems to have been a feasible undertaking as the Graspin environment is designed with the goal to facilitate extensions of its functionality and support the systematic integration of new tools [10].

Independently from this research, we have been adapting Borison's Model of Software Manufacture [2] and the Graph Transform Model for Configuration Management described in Heimbigner and Krane [6] to define a Model of Software Maintenance [14]. This model views maintenance activities as multivalued functions from configurations to configurations. In contrast to our approach, all software objects are considered immutable. Acyclic bipartite graphs are used to model the relations between maintenance steps and the objects in a configuration. To capture hierarchical decompositions of objects, a distinguished is-component-of relationship is imposed on objects. This relationship and the causal structure of maintenance graphs are used to determine induced maintenance steps. Induced maintenance steps are additional steps necessary to preserve the consistency of a configuration after a maintenance charge was applied to an object in a configuration.

The dynamic (temporal) behavior of maintenance models is defined by a local state transition diagram associated with each maintenance step. Each state transition diagram consists of five states: invoked, pending, implementing, completed, and abandoned. They describe the different phases a maintenance step undergoes until it is finally completed or abandoned. The maintenance graph and step states together are used to compute priorities and precedence of maintenance steps for the purpose of task scheduling and decision support.

A basic assumption of this model is that the decision about changes, i.e., about maintenance steps to be taken, is under centralized control. This assumption relieves some of the severe problems which are involved if the decision about maintenance steps to be taken is decentralized, such as concurrency control and configuration consistency. The restriction to centralized decision competence seems adequate for maintenance activities as they are usually performed in large administrations on mainframe computers, for which the model was originally designed. But they fail to meet evolutionary development activities to be observed in the presence of personal computers and loosely coupled networks of dedicated computers.

A future goal of our research in this area is to combine the PNP model with the maintenance model to extend the former by possibilities for computing far-reaching effects of changes and the latter by possibilities to cope with concurrency and to handle the distribution of decision competence.

REFERENCES

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

2. E. Borison, A model of software manufacture, in, *Advanced Programming Environments*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, 1986.
3. B. Curtis, H. Krasner and N. Iscoe, A field study of the software design process for large systems, *Communications of the ACM*, 31, 1268–1287 (1988).
4. T. Colin (ed.), Representing and enacting the software process, *ACM SIGSOFT Software Engineering Notes*, 14, June 1989.
5. A. Finkelstein, ''Not waving but drowning'': Representation schemes for modelling software development, in, *Proceedings of the 11th Annual International Conference on Software Engineering*, Pittsburgh, Pennsylvania, May 1989, pp. 402–404.
6. D. Heimbigner and S. Krane, A Graph transform model for configuration management, in, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1988.
7. 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*, Pittsburgh, Pennsylvania, May 1989, pp. 331–342.
8. B. Krämer, *Concepts, Syntax and Semantics of SEGRAS-A Specification Language for Distributed Systems*, Oldenbourg Verlag, München, Wien, 1989.
9. B. Krämer, Introducing the GRASPIN specification language SEGRAS, In this issue, 1991.
10. B. Krämer and H.-W. Schmidt, Object-oriented development of integrated programming environments with ASDL, *IEEE Software*, January 1989.
11. B. Krämer and H.-W. Schmidt, Architecture and functionality of a specification environment for distributed systems, in, G. Knafl (ed.), *Procs. COMPSAC 90*. Computer Society Press, 1990.
12. Luqi, Software evolution via rapid prototyping, *IEEE Computer*, May 1989, pp. 13–25.
13. Luqi and Y. Lee, Interactive control of prototyping processes, in, *Procs. COMPSAC 89*, Orlando, September 1989.
14. I. Mostov, A model of software maintenance for large scale military systems, Master's thesis, Naval Postgraduate School, Computer Science Department, Monterey, California, 1990.
15. H.-W. Schmidt, *Specification and Correct Implementation of Non-Sequential Systems Combining Abstract Data Types and Petri Nets*, Oldenbourg Verlag, München, Wien, 1989.
16. H.-W. Schmidt, Prototyping and verification of concurrently interacting objects, In this issue, 1991.