1990

Petri net-based models of software engineering processes

Kramer, Bernd; Luqi

Petri Net-Based Models of Software Engineering Processes*

Bernd Krämer and Luqi

Naval Postgraduate School
Computer Science Department
Monterey, CA 93943

Abstract

In this paper a Petri net based formal specification method for distributed systems is accommodated to the application domain of software process modeling. We introduce domain specific concepts stressing the distributedness 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 carriers of such activities, and local changes to development states. These dynamic aspects of software processes are visualized by labeled Petri nets. Structuring mechanisms are sketched which support hierarchical decomposition and systematic combinations of separate views of a software engineering process.

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, 4, 5]. 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 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 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 partially overlapping views of a rapid prototyping process that supports evolutionary software development by interactive construction of executable prototypes from reusable software components [8]. 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.

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

0073-1129/90/0000/0104$01.00 © 1990 IEEE

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 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 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 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 and *structural* properties of objects and information provided at execution time 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 author, interface, body, and creation date attributes. The task 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 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 and 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 accommodating a Petri net based specification method, called *SEGRAS* [6], to the conceptual framework discussed previously. *SEGRAS* was originally designed for writing testing, and analyzing formal specifications of concurrent and distributed software systems. It is based on a well-engineered integration of algebraic specifications and Petri nets. System functions and data objects on which the system operates concurrently are specified as partial abstract data types, while dynamic behavior is presented graphically by means of annotated Petri nets. The PNP model extends this approach 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 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 either are (references to) objects or are 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, if:interface, body:impl)
```

defines objects of type `module` to have at least three 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 `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,[A,I,B,unchkd]>
```

where `unchkd` is such an add-on attribute value which might express the evaluation state of a module.

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 [7]. 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 square brackets denote a list of items of the type they enclose. An example of the second kind is the following:

```
variant eval-state:  (unchkd,ckd,validated: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.

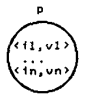
3.2 Process Model Behavior

Objects are created dynamically during process execution. Most of the objects created persist as system development proceeds and simply change their attribute values. 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.

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. 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 terms of the form $\mathtt{a}(\mathtt{X}_1,\ldots,\mathtt{X}_n)$ where `a` is the name of an n-ary action and the $\mathtt{X}_i$ are typed variables whose types match the arity types of `a`. Arcs 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 identity 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. 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.

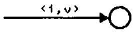
The *marking* of a PNP net is given as a distribution of sets of objects over the S-elements of the net. Markings represent a distributed development state.

A labeled S-element such as

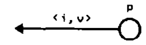


denotes a component of a distributed development state and can be viewed as a variable predicate p whose actual extension is defined by the actual marking and is changed by processes. Labeled arcs express two types of atomic changes:

The form

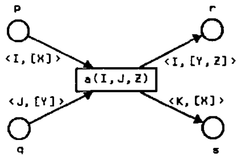


denotes the atomic change that object <id,v> begins to satisfy p, while the form



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

The form



which determines the state changing effect of actions defines a scheme of similar rules of changes by use of variables. We use capitalized words to denote variables, while function, attribute, predicate, and action names are written in lower case. An instance of action is obtained by consistently substituting all variables in the scope of an action (here I,J,K,X,Y and Z) by constants such that the formula, if any, constraining each occurrence of an instance of that action is satisfied according to the specified meaning of functions and predicates the constraint is composed of. Apart from the effect on adjacent S-elements, the above form states that each occurrence of an instance of action a requires the availability external information which consists of the names of existing objects to be operated on (denoted by variable I,K) and additional information necessary to modify their attribute values (denoted by Z).

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 an asterisk (*) or a plus sign (+) to the variable referring to 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, Fig. 1 depicts the behavior of a very simple version control system. This system provides two actions only. Action **establish** 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** 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 allowed to update module m1, m2 and m1, respectively.

To keep the example simple, it gives only an incomplete view of our simple 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 combination mechanisms that allow one to merge simple views in a consistent way to larger and more complex ones. Further we assume the object and data type specifications given in Section 3.1 to be included in the definition part.

In this example we further use a special notation for *mutable side-conditions* of actions by means of dotted arcs. Such side-conditions are just an abbreviation for preconditions of actions that immediately are restored. Here the side-condition expresses the requirement that only authorized authors may perform an update action. Further we use the underscore sign (_) as a wild-match character that matches a whole sublist of attribute values.

3.4 Process Model Dynamics

In a PNP model as shown in Fig. 1 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 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 enabled. Informally speaking, an instance of an action a is enabled in a given development state if the instance satisfies the constraint expression of a (if any), if all objects labeling incoming arcs of that instance are in the marking of the adjacent S-element, and all objects labeling outgoing arcs satisfy the predicate labeling the adjacent S-element. The state change affected by an enabled 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. In [6] a formal definition of these concepts is given for *SEGRAS* nets but similarly applies in a formal framework for PNP models.

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

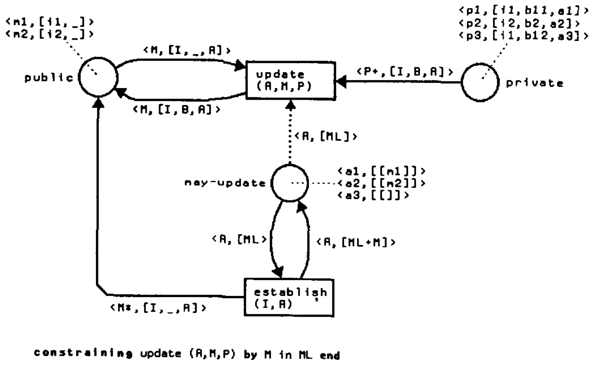


Figure 1: A simple process model controlling the release nd update of of public module public versions

Instances update(a1,m1,p1), update(a2,m2,p2), and update(a3,m1,p3) of action update(A,M,P) in Fig. 1, and many instances of action establish(M,I,A) are enabled.

One of the possible future states of our example is shown in Fig. 2. It was caused by occurrences of update(a1,m1,p1), update(a2,m2,p2), and establish(m3,i3,). 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, update(a1,m1,p1) and update(a3,m1,p3), mutually exclude each other as they "fight" for the same object named m1.

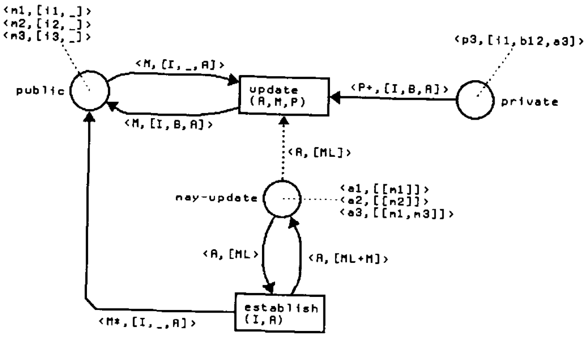


Figure 2: A possible future development state of the process model in Fig 1

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 [8]). 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. Fig. 3 shows a typical example of this informal kind of process models which are often appealingly 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 [9], providing better user guidance, and controlling the application of its tools.

First we define some of the object and data types whose instances are involved in the rules of change specified in Fig. 5 and Fig. 6. The former is a PNP model of Fig. 3. It reveals the nondeterminism hidden in the informal descrip-

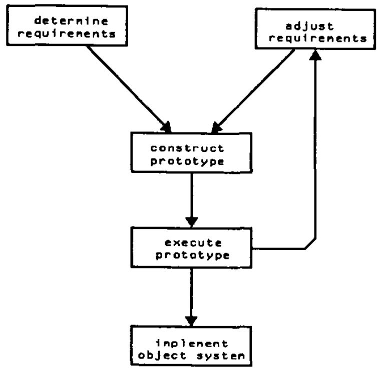


Figure 3: A process model for software evolutionary through prototyping

tion and explicates the information flow necessary to resolve the conflict of whether the execution of a prototype satisfied the users' and developers' expectations and the requirement specifications can be turned into an product version or whether the requirements and the current prototype design must be adjusted and modified using the evaluation protocol, previous requirements and the evaluated prototype as feed-back information. It also shows relations between the various types of documents and how they are updated by development activities. To simplify the graphical presentation of PNP nets, we use the abbreviations depicted in Fig. 4.

The type definitions below refer to software concepts presented in [8]. 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 an control constraints. Operator definitions comprise a name, a specification of input/output data, internal state variables, and constraints, and it possibly comprises an implementation which refines an operator through a data flow network of 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.

```
object req-def:  (sysname:name, description:text)
object operator:  (opname:name, spec:spec)
record spec:  (inputs,outputs,states:[name-type])
record name-type:  (var,type:name))
object impl:  (...)
```

```
predicates requirements(req-def)
           psdl-design(operator)
actions    construct(req-def,spec)
           modify(operator,spec)
           refine(operator,req-def,impl)
           analyze(operator)
vars R:req-def, T:text, S:spec, C:ada-code, ...
```

The process model presented in Fig. 6 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.

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 specification of a newly constructed operator realizing these requirements. We just want to explicate certain relationships concerning names and references among objects. Looking more carefully at the net labelings, we recognize that certain variables denoting attribute vaiues 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 with 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 fixed. The type of variable S determines the structure of the object denoted by S, while the net specifies the behavioral effect.

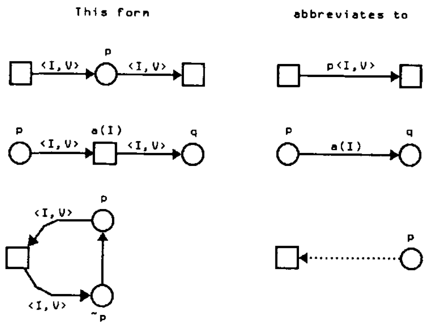


Figure 4: Abbreviations used in PNP models

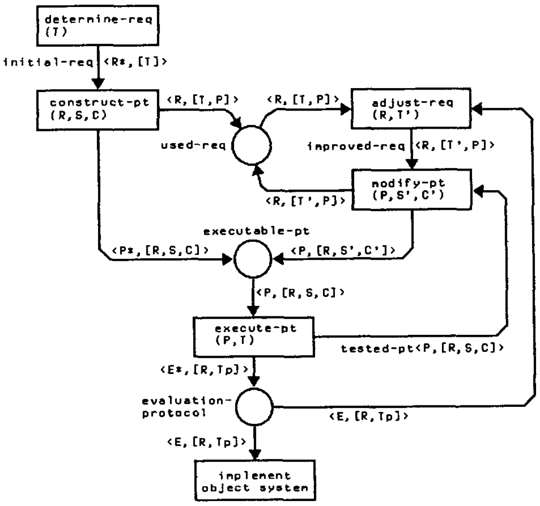


Figure 5: PNP model of the software evolution process model

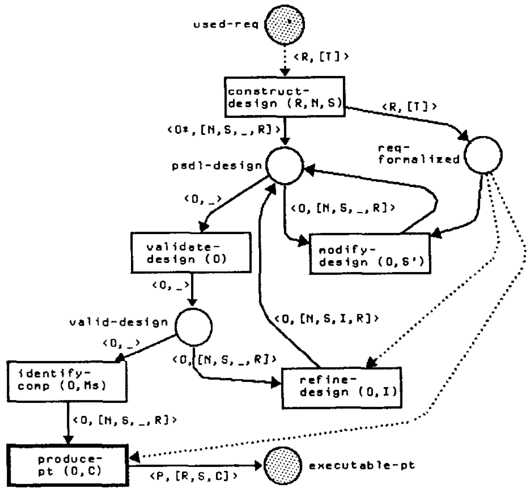


Figure 6: Constructing prototype designs from requirements definitions

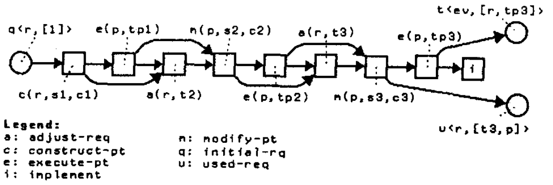


Figure 7: A record of an execution history of the process model shown in 5

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 acyclic 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. Fig. 7 shows a record of the execution history of the PNP model in Fig. 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.

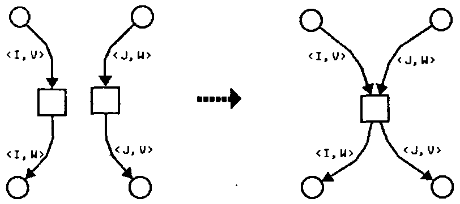
5 Horizontal and Vertical Decomposition of PNP Models

One of the primary difficulties in modeling software processes is conceptual complexity. Conceptual complexity can be reduced if the 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 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 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 constructions and their formal semantics have been developed in the framework of a formal specification language for distributed and concurrent systems [6] 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 marking 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

- 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.

Without mentioning it explicitly, Fig. 6 was a refinement of a subnet of Fig. 5, namely of action construct-pt and its environment. Such refinement and implementation concepts have been studied in [10] for the related specification formalism with particular emphasis on defining suitable correctness criteria which provide the basis for verification tools.

Another example of an action refinement is given in Fig. 8. It shows that action produce-prototype, which appeared in Fig. 6, 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. 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 of such a design to implement the communication interfaces MI between reusable components implementing operator specifications.

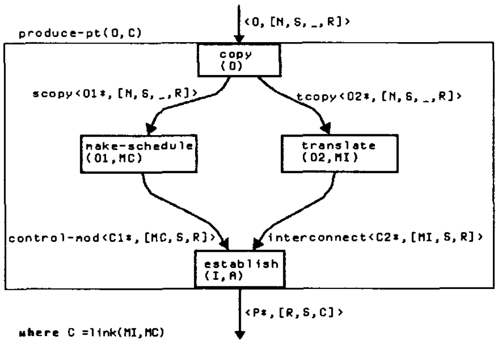


Figure 8: Refining an action of the process model in Fig. 6

A scheduler attempts to produce a feasible schedule MC for the execution of time critical operators. (The possibility that this attempt may fail is not shown in our example.) The generated Ada packages MI, MC then are linked together with the reused components Ms to form the executable prototype. The meaning of the link function and the type definitions for Ada components are not given here but are straightforward.

6 Conclusions

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. 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 term 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 [6]. This has the advantage that the functions of construction and analysis tool [2] that have been developed for *SEGRAS* can be lifted to the description level of PNP models. Currently, the *SEGRAS* 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 graphical representation of nets,
- liveness and safeness analysis for restricted class of *SEGRAS* nets,
- and verification of algebraiç specifications.

Moreover, due to the restrictions we have sketched concerning object identities and manipulation, our nets seem to have the properties that would make the whole class of PNP nets amenable to liveness and safeness analysis developed in [10]. 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 subnets 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. 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.

Animation of PNP models through symbolic execution 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.

Further extensions of the PNP model could attempt to exploit the area of performance analysis on the basis of timed Petri nets for the purpose of cost estimation.

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] 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.

[6] Bernd Krämer. *Concepts, Syntax and Semantics of SEGRAS - A Specification Language for Distributed Systems.* GMD-Bericht Nr. 179. Oldenbourg Verlag, München, Wien, 1989.

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

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

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

[10] 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.