A Flexible and Formalized Process Modeling Language*

CHEN Cheng[1], SHEN Bei-jun[1,2], GU Yu-qing[1]

[1](*Institute of Software*, *The Chinese Academy of Sciences*, *Beijing* 100080, *China*);

[2](*Department of Computer Science*, *East China University of Science and Technology*, *Shanghai* 200237, *China*)

E-mail: chen_goodwin@yahoo.com

Received July 18, 2001; accepted January 14, 2002

Abstract: For reaching the requirement of process domain, a flexible and formalized process modeling language FLEX is proposed to support semantics richness, easy of use, flexibility, scalability, reuse, and distribution, while it is analyzable, executable, and evolutive. Especially, the language not only can provide nonexperts high level representation for easy of use, but also can allow users to define and reuse process notations at various granularities to extend the representation. So FLEX can support various levels and requirements of process modeling.

Key words: software process; process modeling; object oriented; pattern; PSEE

Software process is all the real-world elements involved in the development and maintenance of a software product, i.e. resources, activities, artifacts and organization[1]. A formalized process model should be specified to support Process-centered Software Engineering Environment (PSEE) or Workflow Management System (WfMS). Through investigating many process modeling languages (PMLs), focusing on second generation PMLs[2] proposed since 1996, we found that PML should support semantics richness, easy of use, flexibility, scalability, reuse, and distribution, while it should be analyzable, executable, and evolutive.

It's obvious that current PMLs and their support systems can't reach those requirements. Most of PMLs[2~6] can only specify and execute process model at low level abstraction, i.e. petri nets, rule-based formalism, and procedure languages. Although some of them, such as JIL[2], SPADE[3], MARVEL[4], use graphical representation to make process model more comprehensible, the granularity of process model is superfine to impede understanding and reuse. Thus, a high level PML is needed, which is intuitive enough for nonexperts to specify problem domain. Object-oriented modeling approach[7,8] seems to be suitable for the requirement, because it provides uniform and powerful representation capabilities for the different aspects of a process since they rely on a natural way of identifying and encapsulating existing entities. But it has disadvantages that it hasn't definite executable semantics and no global functional and behavior view of process model exists. In recent years, APEL[9] and MOKASSIN[10, 11] try to provide users high level formalism, while supporting process execution by compiling the graphical representation into executable formalism, but their translators are pre-defined. For more flexibility, a PML should support the user-adaptable informal representation and the approach to transform gradually an informal model into a

* Supported by the Innovation Foundation of Institute of Software, The Chinese Academy of Sciences (ISCAS) under Grant No.CX2K5415 (中国科学院软件研究所创新基金)

CHEN Cheng was born in 1978. He is a Ph.D. candidate at ISCAS. He received his B.S. degree in computer science from USTC in 1997. His research interests are software engineering, software process and software architecture. **SHEN Bei-jun** was born in 1969. She is a Ph.D. candidate at ISCAS, and a lecturer in East China University of Science and Technology. Her research interests are software engineering, software process, software tools and enterprise network computing. **GU Yu-qing** was born in 1940. He is a professor and doctoral supervisor at ISCAS. His current research areas include software engineering.

formal one[9], but none of existing PMLs can reach the requirement.

We propose a process modeling language FLEX that can support all features mentioned above. Based on object-oriented, rule-based and constraint-based techniques, FLEX provides an abstraction mechanism that can not only provide nonexperts high level representation for easy of use, but also allow users to specify process model at different granularities for both semantics richness and flexibility. Typically, User can also reuse existing process notations at different abstraction levels, based on their knowledge about the semantics, to construct higher level notations to extend its expressive power. Only experts need to cope with lowest level representation. Moreover, the formalized process model in FLEX can be analyzed for keeping the consistency, and can be executed and evolved in FLEX support system.

In this paper, we focus on introducing the specification method of FLEX. The approach for analysis and evolution will be mentioned in forthcoming papers. In Section 1, we identify architecture and main features of FLEX support system. Section 2 briefs the executable and analyzable sub-language FLEX/BM. Section 3 introduces the abstraction mechanism with the explanation of how to construct process elements, control flow, data flow, and etc. In Section 4, we assess our approach and give a conclusion.

1 Architecture and Features of FLEX Support System

The language FLEX has two representations, one is the pre-defined high level graphical representation FLEX/PL, and the other is the executable and analyzable representation FLEX/BM. It shows the architecture of its support system that consists of graphical editor of FLEX/PL, abstraction mechanism for user-defined notations, a FLEX language transformer, a process analyzer, a process interpreter, a process engine, a process monitor, a process controller, and a process evolution manager in Fig.1.

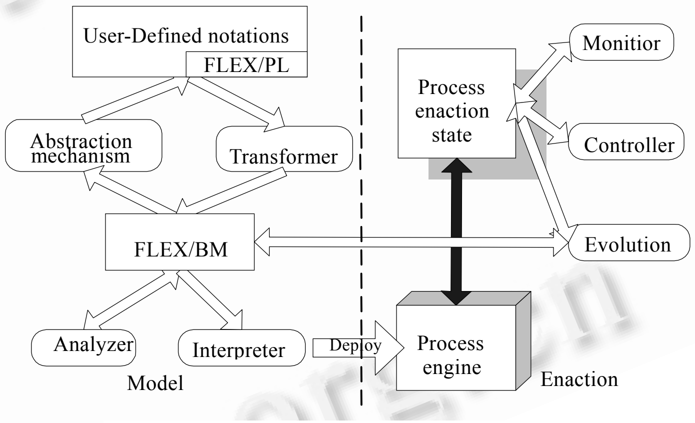


Fig.1 Architecture of FLEX support system

FLEX support system can meet the following goals for supporting variable process scenarios.

- *semantics richness*, FLEX/BM is an executable language, which can specify process model with fine details,
- *easy of use*, FLEX/PL provides high level representation which suits to nonexperts, and the process model and its elements can be refined step by step into hierarchy structure,
- *flexibility, scalability and reuse*, particular abstraction mechanism of FLEX enables user-defined notations on the basis of FLEX/BM to extend FLEX/PL and to reuse process fragments,
- *distribution*, the basic objects communicate with each other by message passing, process engine provides the support of passing messages and enable the distributed process model,
- *analyzable*, on the basis of finite state verification (FSV)[12,13], the consistency, no deadlock, and other properties of a process model in the form of FLEX/BM can be analyzed,
- *evolutive*, FLEX support system not only can evolve process models, but also can evolve itself.

2 Overview of FLEX/BM

In FLEX/BM, a process model can be regarded as components that may execute in concurrent way, with some patterns that constraint the behavior of the process model. The component, i.e., object encapsulates some user-defined data and provides some operations. An object can not access other objects directly, can but communicate with other objects by message-passing mechanism. Patterns specify the needful properties of operations' occurring order while executing the process model, so we call it as pattern constraint later. The execution rule of the process model in FLEX/BM is as follows: while an object receives an event, it executes the corresponding operation if the operation doesn't conflict with all pattern constraints of the process model, otherwise the operation should be rejected.

2.1 Object

Like common object-oriented systems, FLEX/BM has some built-in objects, such as String, Numeral, Boolean, and Set. User can construct objects by three built-in relations: aggregation, generalization, and association.

Both event and condition can trigger an operation of object. Commonly, it can be specified as an ECA rule[11] in the form of "ON event IF condition DO action", which shows that an action should be performed if the specified condition is satisfied while an event occurs. FLEX/BM prescribes the operations in one object are serialized. Namely, in one object, only one operation can be performed at any time, and events occurring while an operation is executing will be performed after the operation is finished. It makes the semantics of those operations in concurrent objects can be characterized by interleaving.

2.2 Pattern constraint

The operation sequence while executing a process model should satisfy all of the pattern constraints in the process model, that is a regular expression whose operands are operations of process element and operators can be *subsequence* (;), *concurrence* (||), *exclusive OR* (⊕), *NOT* (¬), *optional* (**[]**), *iterative* (+), and *optional iteration* (*). With the quantifier ∀ and ∃, we can specify the pattern constraint on the operations of a kind of objects. Our pattern constraint derives from the idea of the operation pattern in OBM[14], and provides more powerful and intuitive representation. Firstly, there are only a part of operations in one pattern constraint, so any operations that haven't been mentioned can execute in any order. In addition, except for specifying pattern constraints on the operations of an object, FLEX/BM allows to specify pattern constraints on the operations of multiple objects.

3 Abstraction Mechanism of FLEX

FLEX support to construct notations for specifying process model to extend itself with the abstraction mechanism on the basis of FLEX/BM, so the high level representation FLEX/PL.

3.1 Common process elements in FLEX/PL

Although existing process modeling languages have various notations and formalisms, there are some acknowledged process elements, such as *activity*, *product*, *role*, *agent*, and *tool*. In FLEX/PL these process elements are pre-defined objects with definite semantics, and users can define new process elements by the built-in relations in FLEX/BM. The graphical notations of those elements are shown in the following:

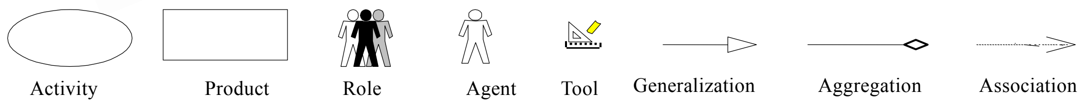
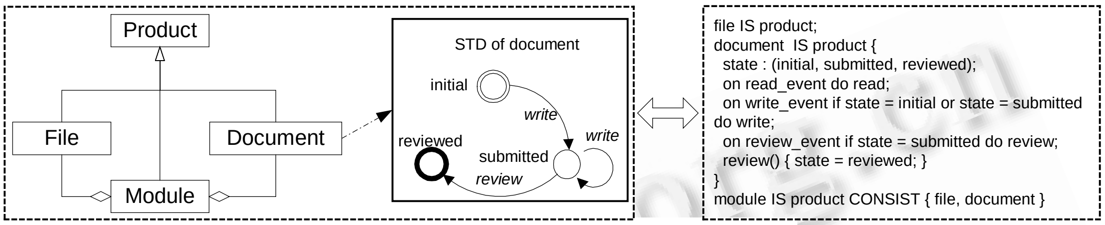


Fig.2 Graphical notations of some process elements and relations

For example, the *product* object has two fundamental operations *read* and *write*, and two states *initial* and *submitted* whose transitions are specified by *state transition diagram*. An example shows three user-defined products and their structures in the left section of following figure. There some files and documents constitute a module, and documents are special products that should be reviewed to ensure their quality. So the *module* can be specified as the aggregation of the *file* and the *document*. New state *reviewed*, *operation review*, and modified state transition diagram are added to the specification of the *document*. The specification of products in the left section is high level and intuitive, which is the abstraction of the FLEX/BM program in the right section, where the definition of *product* object is omitted.



```
file IS product;
document  IS product {
  state : (initial, submitted, reviewed);
  on read_event do read;
  on write_event if state = initial or state = submitted
do write;
  on review_event if state = submitted do review;
  review() { state = reviewed; }
}
module IS product CONSIST { file, document }
```

Fig.3 An example for constructing products

There are some operations in the *activity* object, that are *start*, *get*, *submit*, *suspend*, *resume*, *commit* and *abort*, where the get and submit are operations getting and submitting a product by calling the *read* or *write* operation of the product, others are operations controlling the execution of activity. The behavior of them is restricted by such pattern constraint $\text{PAT}_{\text{activity}}$ "*start* ; ((*get** || *submit**) ; (*suspend* ; *resume*)*)* ; (*commit* ⊕ *abort*)". It implies that an activity should be started firstly, whereafter it can get or submit some products, meanwhile it may be suspended and resumed, and it can be committed to finish normally, or be aborted to finish abnormally. While an enacting activity is suspended, only the operation *resume* can be executed to continue the enaction.

To the *role* object, we focus on its *skill requirement* attribute, which is a set of skill type and degree. Only those agents who reach the skill requirement of a role can be assigned with the responsibility of the role.

3.2 Advanced control flow specification method

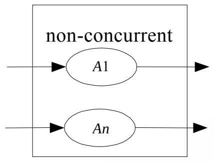
On the view of ontology, the relations are more complex and important than the objects in a system. Therefore, the specification of a process model should stress on the relations between process elements. The most important relations are the relations related to the *control flow* and the *data flow*. The control flow is the rules to restrain the execution sequence of operations in process model. In FLEX/BM, the control flow can be specified in flexible manner using pattern constraint, because it supports to specify the execution order of operations in detail. Here we will show the construction rules of some pre-defined control flow notations in FLEX/PL on the basis of FLEX/BM, and users can construct user-defined notations for specifying control flow in similar way.

Firstly, an activity can have different *instantiation mark*, which decides the possible behavior of the activity while it's instantiated. Reference to [1], instantiation mark of activity consists of *optional*, *no reactive*, *serial*, and *periodic*. In FLEX, the instantiation properties of activities can be transformed into pattern constraints. An *optional* activity A should have the pattern constraint (*instantiate*(A))*, which shows that the optional activity can be skipped. A *no-reactive* activity A should have the pattern constraint [*instantiate*(A)], which shows that the activity can be instantiated once at most. A *serial* activity A should have the pattern constraint $\forall(i)$ ($\text{PAT}_{\text{activity}}(A(i))$; [*instantiate*($A$)]), which shows that only after current instance of A is finished, the activity can be instantiated once more. A *periodic* activity A should have the pattern constraint (PeriodTimer; *instantiate*(A))*, where PeriodTimer is an operation of timer that executes periodically. The pattern constraint shows that the activity should execute periodically after executing the operation PeriodTimer.

Activities in a process model will be executed concurrently if without any constraints. In most circumstance, some constraints on the execution order of activities should be specified in order to reduce the indetermination of the behavior of the process model. For example, there are some typical relations between two activities below that decide the behavior of activity B based on the execution status of activity A, which can be transformed into pattern constraint specifications:

Table 1 Relations between two activities and their pattern constraint

Relation			Description	Pattern constraint
	finish-start		B can start only after A is finished.	(A.commit ; [B.start])*
	start-start		B can start only after A is started.	(A.start ; [B.start])*
	start-finish		B can finish only after A is started.	(A.start ; [B.commit])*
A	finish-finish	B	B can finish only after A is finished.	(A.commit ; [B.commit])*
	after-expect		After A is finished, B should be executed.	(A.commit ; B.start)*
	after-prohibit		After A is finished, B can't be executed.	(A.commit ; ¬ B.start)*
	while-prohibit		While A is executing, B can't be executed.	(A.start ; ¬ B.start ; (A.commit ⊕ A.abort))*



Obviously the relations between two activities can be readily extended to form the relations among multiple activities to determine the execution sequence of activities. But sometimes user needs to specify some activities to be non-concurrent, i.e., only one of the activities can be executed at any time. In this situation, the execution sequence of these activities isn't determinate, and process performer can select and execute one of them at one time.

For reaching the requirement, relation *non-concurrent* among activities is introduced, which can be transformed into a pattern constraint $\mathrm{PAT}_{\text{non-concurrent}}=\forall(A,B|A,B\in S)$ (A *while-prohibit* B), where S is the set of those non-concurrent activities $\{A_1,...,A_n\}$. The pattern constraint shows that while executing any activity in S, another activity can't be executed. So the execution order of activities in S can only be serial, but the pattern constraint doesn't constrain the execution behavior of those activities, which are determined by the pattern constraints of each activity itself.

3.3 Data flow specification method and collaboration mechanism

In process model, the input and output products of activities construct the data flow, and determine the permissibility of the activity operates the products. Obviously, an activity can read and write its output products. In FLEX/PL, an activity can access its input products in *read only* mode or *write enable* mode.

If an activity A can only read its input product P, a pattern constraint "$\neg$ *A.submit*(P)" should be satisfied. On the other hand, if a product P can be written by multiple activities $A_1,A_2,\ldots,A_n$, a synchronization mechanism should be used to keep the consistent version of the product. User can define version control mechanism, and there are two pre-defined mechanisms in FLEX/PL. One is Multi-Version Concurrency Control (MVCC) Mechanism[15]. After a product is changed by an activity, other activities should get the newest version of the product before attempting to submit it. The mechanism can be presented in the following pattern constraint:

$$\forall(A|A\in\{A_1,A_2,\ldots,A_n\})(A.submit(P)\ \textit{after-expect}((\backslash A).get(P);\ [(\backslash A).submit(P)]))$$

The other is check-in/check-out mechanism, where only the activity that checks out a product can check in (submit) the product. If user chooses to use the check-in/check-out mechanism, the operations of activity to operate products are changed to *get* (*read*), *check-out* (*read*), and *check-in* (*write*). These three operations satisfy the following pattern constraint:

$$\forall(A|A\in\{A_1,A_2,\ldots,A_n\})(A.\textit{check-out}(P)\ \textit{before-prohibit}\ A.\textit{check-in}(P))\ \text{and}\ (A.\textit{check-out}(P)\ \textit{after-prohibit}\ (\backslash A).\textit{check-out}(P))$$

Product change control mechanism in the collaboration of multiple activities can also be described in pattern constraints. If an input product of an activity is changed by other activities, the activity should get the current version of the modified input product before it tries to submit some output product. The corresponding pattern constraint is "([*P.write*] ; *A.get*(*P*) ; ∀(*o* | *o* IS product) [*A.submit*(*o*)])*", where *P* is an input product of *A*.

3.4 A simple process model example

Here we use FLEX representation to model a simple process to exemplify some benefits of our language in Fig.4. It consists of three concurrent activities, which are *design_step, coding_step* and *review*, where the result of executing activity *design_step* is *design_document* that should be went through by activity *review*. If the *design_document* can't reach the expected requirement, a *feedback* message that is a special product will be sent to activity *design_step* for requiring a revision, otherwise a *review_report* will be submitted. Only after the *design_document* has past the review, activity *coding_step* can proceed and generate *source_code*. Data flow, control flow, and mechanism of product control and collaboration of the process model are implied in pre-defined relations so that the process model looks concise and intuitionistic, and two explicit textual sentences are specified for customizing the process model.

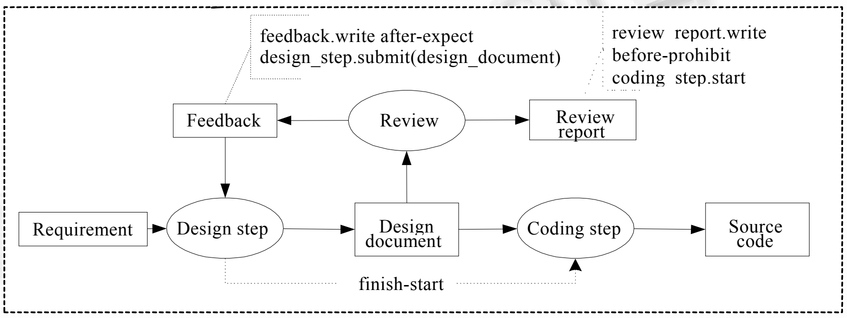


Fig.4 Main part of a simple process model example

In this example, data flow relations in the process model imply the mechanism of product access and change control, which can be transformed into some pattern constraints according to the abstraction mechanism,

4 Conclusion and Future Work

We propose a flexible and formalized process modeling language FLEX that bases on object-oriented, rule-based and constraint-based techniques. FLEX and its support system can reach the requirements in process domain, such as semantics richness, easy of use, flexibility, scalability, reuse, and distribution. Two sub-languages, FLEX/PL and FLEX/BM, are proposed to different goals. FLEX/PL aims to be easy of use for non-experts, while FLEX/BM aims to definite and rich semantics. In this paper, we focus on introducing the abstraction mechanism of FLEX, which supports to define high level notations on the basis of FLEX/BM in order to glue the gap between the notations at different abstraction levels. The features of FLEX/PL in detail are not involved in this paper, which can be found in Ref.[16].

The most important advantage of the abstraction mechanism of FLEX is that it supports users (not only experts) to customize the notations for special requirements in various granularities based on existing notations. And the abstraction mechanism of FLEX supports to transform gradually the user-defined informal representation into a formal FLEX/BM representation. In this paper, clearly there are specification methods at four different abstraction levels at least.

(1) *Implement level*: The method at lowest abstraction level is the object specification in FLEX/BM that can implement all functions of a process model needs, which is similar to the formalism of MOKASSIN that are based on rule-based formalism. Only experts can cope with the representation of this level.

(2) *Constraint level*: The pattern constraint can specify the behavior of process model intuitively. In this level,

the properties, not the procedures, of process model are specified. Especially, the properties involving multiple objects can be specified directly.

(3) *Semi-nature level*: For example, the temporal relations among operations can specify the temporal order of operations in a way that is similar to natural language description, such as “after ... should ...”. Most of users can readily understand it without different meanings.

(4) *Graphical level*: The graphical notations, both pre-defined and user-defined, is the high level notations that can be readily understandable. Different organizations or users can customize their own graphical notations.

In contrast, APEL has rich expressive power by defining abundant graphical process notations, which cover most of process elements, control flow, data flow, state diagram, concurrence, and collaboration. MOKASSIN pays attention to the process modeling in workflow. It integrates the high level constructs of task graphs and the flexibility of rule-based techniques into a coherent framework, hence can support the user-adaptable and flexible process modeling. But, both APEL and MOKASSIN can only construct user-defined notations or customize process model by specifying rules. The method is difficult to most of users because it involves too many details in process model. In addition, the user-customized rules must influence other parts of the process model, so the rule-based process model will become complex and uncontrollable. Hence, our approach with abstraction mechanism is more flexible, intuitive, and easy to specify process model and to reuse existing process notations.

Acknowledgement Our work is on the basis of E-Process, which is a commercial web-based distributed PSEE funded by SRA and ASTI. Here we would like to thank all members of E-Process project.

References:

[1] Derniame, J.C., Kaba, B.A., Wastell, D. Software Process: Principles, Methodology and Technology. Springer Verlag, 1999.

[2] Sutton, Jr., S.M., Osterweil, L.J. The design of a next-generation process language. In: Proceedings of the 6th European Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Software Engineering. Springer-Verlag, 1997. 142~158.

[3] Bandinelli, S., *et al*. SPADE: an environment for software process analysis, design, and enactment. In: Software Process Modeling and Technology. Research Studies Press Ltd., 1994. 223~247.

[4] Kaiser, G.E. MARVEL 3.1: a multi-user software development environment. In: Proceedings of the International Symposium on Logic Programming. Vancouver, Canada, 1993.

[5] Rombach, H.D. MVP-L: a language for process modeling in-the-large. Technical Report UMIACS-TR-91-96, University of Maryland, 1991.

[6] Canals, G., *et al.* ALF: a framework for building process-centred software engineering environments. In: Software Process Modeling and Technology. Research Studies Press Ltd., 1994. 153~185.

[7] Baldim, M., *et al*. Object oriented software process model design in E3. In: Software Process Modeling and Technology. Research Studies Press Ltd., 1994. 279~290.

[8] Rumbaugh, J., Jacobson, I., Booch, G. The UML Reference Manual. Addison Wesley, 1999.

[9] Dami, S., *et al*. APEL: a graphical yet executable formalism for process modeling. In: Automated Software Engineering (ASE). 1997.

[10] Joeris, G., *et al*. Towards object-oriented modeling and enacting of processes. TZI-Report 07/98, Center for Computing Technologies, University of Bremen, 1998.

[11] Joeris, G. *et al*. Towards flexible and high-level modeling and enacting of processes. In: Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CaiSE’99). 1999.

[12] Holzmann, G.J. The model checker SPIN. IEEE Transactions on Software Engineering, 1997,23(5):279~295.

[13] Cobleighm, J.M., *et al*. FLAVERS: a finite state verification technique for software systems. Technical Report, UM-CS-2001-017, Department of Computer Science, University of Massachusetts, Amherst, MA 01003. 2001.

[14] Sa, J., *et al*. OBM: a specification method for modeling organizational process. In: Proceedings of the Workshop on Constraint Processing at CSAM’93. 1993.

[15] Multiversion Concurrency Control. 2002. http://www.postgresql.org/idocs/index.php?mvcc.html.

[16] Chen, Cheng, Shen, Bei-jun. Towards flexible and high-level process modeling language. In: Proceedings of the International Symposium on Future Software Technology (ISFST). 2001. 136~141.

一个柔性的形式化过程建模语言

陈 诚[1], 沈备军[1,2], 顾毓清[1]

[1](中国科学院 软件研究所,北京 100080);

[2](华东理工大学 计算机科学系,上海 200237)

摘要: 提出了一个柔性的形式化过程建模语言 FLEX.它具有丰富的语义、易用性、灵活性、可扩充性、可重用性和分布性,是一个可分析、可执行、演化的过程建模语言.建模语言不仅提供了高度抽象的描述方法以便于一般用户使用,还允许用户通过重用的方式自定义多种抽象级别的语言元素来扩充语言的描述能力.因此,FLEX 语言可以支持不同层次和需求的过程建模.

关键词: 软件过程;过程建模;面向对象;模式;PSEE

中图法分类号: TP311 文献标识码: A

◇◇

第 **12** 届中国计算机学会网络与数据通信学术会议

征 文 通 知

为推动我国在此方向的研究，探讨计算机网络与数据通信技术的发展动态与趋势，促进我国科研人员在此领域的交流与合作,中国计算机学会网络与数据通信专业委员会拟于 2002 年 12 月 2 日~4 日在武汉举办“ 第 12 届中国计算机学会网络与数据通信学术会议 ”。会议由华中师范大学计算机科学系承办，并将邀请该领域的国际知名学者作专题特邀报告。为保证本次会议的学术质量，现向全国科技工作者公开征稿。征稿范围包括计算机通讯网络理论与工程的各个方面。本次会议的论文将结辑出版优秀论文将由计算机学会推荐给有关核心期刊发表。

征文要求：

1.论文应是未公开发表过，一般不超过 6 千字

2.全文电子邮件投稿，要求 Word2000 兼容的电子文档，所有内容放于一个文件中

3.编排格式

标题：居中，2 号黑体, 作者：居中，4 号仿宋, 作者地址：5 号楷体

摘要、关键词：5 号楷体 正文：5 号宋体，分节标题 4 号

参考文献：小 5 号宋体

4.投稿地址：华中师范大计算机科学系 谭连生 教授收 (E-mail: L.Tan@ccnu.edu.cn)

重要日期：论文提交截止日期：2002 年 8 月 15 日 论文接收通知日期：2002 年 10 月 1 日

会议注册日期： 2002 年 12 月 2 日

联系方式：

联系人： 谭连生教授, 电 话：027̄87673277, 传 真：027̄87876070, E-mail: L.Tan@ccnu.edu.cn

通信地址：湖北省武汉市华中师范大学计算机科学系, 邮 编：430079