A Goal-Based Approach for Process Instance Evolution[*]

CHEN Cheng, GU Yu-qing

(*Institute of Software*, *The Chinese Academy of Sciences*, *Beijing* 100080, *China*)

E-mail: chen_goodwin@yahoo.com

Received September 7, 2001; accepted January 31, 2002

Abstract: Process is long-lived, distributed, heterogeneous, and always evolutive to cope with unforeseen environment. In this paper, an approach for process instance evolution is proposed, which is a formal approach that focuses on ensuring the "goal-based behavior consistency" between the evolved process instance and the original instance in order to avoid ad-hoc change of process model. The goal is regarded as the criterion of behavior consistency, so evolved process instance needs to be analyzed in formal methods to ensure that it can still achieve its goals. Through specifying the semantics of process model and goals as Büchi automaton, whether an executing process instance can achieve its goals or not is decided.

Key words: process instance; process model; evolution; goal-based

Process is investigated for software development, project management, business management, quality assurance, and etc. Generally a formalized process model should be specified and enacted in Process-centered Software Engineering Environment (PSEE) or Workflow Management System (WfMS); both of them are called Process Support System (PSS). Because process enaction is long-duration and in highly dynamic environment, actual executing process may deviate from specified process model because of unforeseen situations, such as change of user requirement, technical problems, and dismission of some employees. For handling inconsistency between process model and actual process, some operations within meta-processes, that are special processes, should be performed to evolve process model, i.e., to change process model in order to adapt the environment.

Process evolution covers various problems from making minor changes while executing process model, to large-scale changes such as applying new software development standards. Some existing PSSs provide the support of process evolution. For example, SPADE[1] can specify the mechanism of process evolution as a meta-process model with its reflective process modeling language SLANG and enact it. There are two modes in SPADE to impact the change of process model on executing process instance, one is *eager* mode that changes executing process instance immediately, and the other is lazy mode that only impacts on new process instances, except current process instances. Obviously eager mode is more difficult to be implemented but effective than lazy mode, and most of the mechanisms of process evolution focus on investigating the eager mode, such as PIE[2]. Because impacting change on the executing process instance immediately, the eager mode is also called process instance evolution. Additionally, the evolution of process model is necessary only if the deviation will affect new process instances, but in many situations the deviation from process model is minor and temporary, so only process instances should be evolved. The evolution that only affects process instance has been regarded as flexible process enaction mechanism

∗ 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. **GU Yu-qing** was born in 1940. He is a professor and doctoral supervisor at ISCAS. His current research areas include software engineering.

in some workflow systems[3].

Current PSSs put most of their efforts on the strategy of process evolution, but few of them care for consistency between original process instance and evolved process instance. Here consistency implies that some specified properties of a process model, such as the goals, should be satisfied after evolution. Intuitively, a consistent process evolution operation can avoid ad-hoc change to the process model. In Ref.[4], we have proposed a flexible process modeling language FLEX and its support system, which consists of tools that support analysis, enaction, monitor, and evolution. On the basis of FLEX, we'll propose a goal-based process instance evolution approach with consistency assurance, focusing on the invariance of functional goals of a process instance during evolution.

In Section 1, we briefly introduce the basic semantics of our process modeling language FLEX. Section 2 introduces our goal-based approach for process instance evolution. Section 3 constructs finite state automaton according to the semantics of a process model as part of the proof in Section 2. Section 4 gives a conclusion of our process instance evolution approach.

1 Overview of FLEX

FLEX is a process modeling language that can reach the requirements of semantics richness, easy of use, flexibility, scalability, reusability, and distribution in process domain. And it supports users to define new process notations at various granularities to extend the representation, and transforms the user-defined notations into low-level form for analysis and enaction. The detail introduction of FLEX can be found in Ref.[4].

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. In FLEX/BM, a process model can be regarded as objects that may execute in concurrent way, with some patterns that constrain their behavior. The object can communicate with each other by message passing. 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 in FLEX/BM is as follows: while an object receives an event, it executes the corresponding operation if the operation doesn't conflict with any pattern constraints of the process model, otherwise the operation should be rejected.

We focus on the semantics of pattern constraint. We denote that an operation sequence can satisfy a pattern constraint if it doesn't conflict with the constraint. Obviously, there is a set of operation sequences that can satisfy a specified pattern constraint, and the actual execution sequence of operations should be in intersection of the sets that corresponding to all pattern constraints in a process model. Formally, assuming a finite nonempty set of operations to be Σ, and an infinite operation sequence be an element of Σ^{ω}, we define $L_{\omega}(\Sigma,P)$ to be the set of operation sequences that satisfy pattern constraint P over the alphabet Σ. In most situation, we define Σ to be the set of all operations in the process model, and abbreviate $L_{\omega}(\Sigma,P)$ to $L_{\omega}(P)$. Hence, the atomic pattern constraint, operation O, can be satisfied by $L_{\omega}(O)=\{a^{*}Oa^{\omega}|a\in\Sigma-\{O\}\}$. And complex pattern constraints can be constructed by the following rules:

- $\boldsymbol{P}_1 \,||\, \boldsymbol{P}_2$ is satisfied while both P_1 and P_2 are satisfied, i.e., $L_{\omega}(P_1||P_2)=L_{\omega}(P_1)\cap L_{\omega}(P_2)$,
- $\neg \boldsymbol{P}$ is satisfied while P isn't satisfied, i.e., $L_{\omega}(\neg P) =\Sigma^{\omega}-L_{\omega}(P)$,
- $\boldsymbol{P}_1\vee\boldsymbol{P}_2$ is satisfied while one of P_1 and P_2 is satisfied, i.e., $L_{\omega}(P_1||P_2) =L_{\omega}(P_1)\cup L_{\omega}(P_2)$,
- $[\boldsymbol{P}]$ is satisfied while P is satisfied, or none of the operations in P hasn't been executed, i.e., $L_{\omega}([P])=L_{\omega}(P)\cup\{\varnothing\}$, where $\varnothing$ is a null operation,
- $\boldsymbol{P}^{+}$ is satisfied while P is satisfied, or P is satisfied again, i.e., $L_{\omega}(P^{+}) = \{a^{\omega} \mid a \in L_{\omega}(P)\}$,
- $\boldsymbol{P}*$ equals to pattern constraint $[P^{+}]$,
- $\boldsymbol{P}_1;\boldsymbol{P}_2$ is satisfied while P_1 and P_2 are satisfied in proper sequence, i.e., $L_{\omega}(P_1;P_2)=\{ab|a\in L_{\omega}((\Sigma-\Sigma_{P2}\cup\Sigma_{P1})$,

$P_1),b\in L_\omega((\Sigma-\Sigma_{P1}\cup\Sigma_{P2}),P_2)\}$, where Σ_{P1} and Σ_{P2} are the sets of operations specified in pattern constraints P_1 and P_2.

Pattern constraint derives from the idea of the operation pattern in OBM[5], and provides more flexible and intuitive representation. User can define a part of operations in one pattern constraint for convenience, and any operations that haven't been mentioned can execute in any order. In addition, except for specifying pattern constraints over the operations of an object, FLEX/BM allows to specify pattern constraints over the operations of multiple objects, in order to specify global behavior directly and intuitively.

2 A Goal-Based Approach for Process Instance Evolution

2.1 Process instance evolution primitives

At first, we propose some process instance evolution primitives in FLEX to evolve process model during enaction and impact on executing process instance immediately, and they can be categorized into object and pattern constraint evolution primitives.

• object evolution primitives can change the concurrent objects in process model, which consist of (1) create an new object, (2) remove an object, (3) create an attribute, (4) modify the type of an attribute, (5) remove an attribute, (6) create an operation, (7) modify an operation, (8) remove an operation.

• pattern constraint evolution primitives can modify the pattern constraints in process model, which consist of *create*, *enable* and *disable*. User can create pattern constraints in process model, and the execution of process model must satisfy an enabled pattern constraint, and mustn't satisfy a disabled pattern constraint. In addition, some primitives can be used to modify a pattern constraint, which are (1) $P\rightarrow P||P'$ (where $\rightarrow$ denotes '*is modified to*'), (2) $P\rightarrow P;P'$, (3) $P\rightarrow P';P$, (4) $P\rightarrow P\vee P'$, (5) $P\rightarrow\neg P$, (6) $P\rightarrow[P]$, (7) $P\rightarrow P^+$, (8) $P\rightarrow P^*$, and their contradictories.

We can simply analyze the impact of those evolution primitives on the static structure of modified process model. Firstly, creating an object or attribute doesn't break structure consistency of a process model because the new object or attribute is isolated. Secondly, an operation can be removed only while no objects call the operation, and an attribute can be removed only while no operations access the attribute. Removing an object can be regarded as the combination of removing its operations and attributes. Thirdly, modifying the type of an attribute should ensure that all operations access the attribute can also access the modified attribute. Creating or modifying an operation should also ensure the operation to be able to access involved attributes, and to call other operations in legal way. Lastly, change of pattern constraints can keep structure consistency of the process model if the modified pattern constraints satisfy their syntax.

2.2 Goal-Based behavior consistency

Although the correctness of static structure of a process model can be checked to avoid ad-hoc change, sometimes it's more important to ensure the behavior of evolved process instance is reasonable. Hence, our goal-based approach regards the goals as the relation between original and evolved process instances, and defines that the evolution can keep the behavior consistency if the goals of original process instance can also be achieved by evolved process instance. Goals of a process model consist of (1) functional goals, which specify functions that the process model promises to perform, i.e., the satisfied properties of products, (2) time and resource constraint goals, which specify the assigned duration and resources for executing the process model. There should be at least an execution sequence of the process instance that can achieve the functional goals, while satisfying the required time and resource constraint goals in the process model. In this paper, we only care about the functional goals in a process model.

The functional goals of process model can be specified in the form of predicate logic with equal notation. They can also be united to one goal through the union operator. Because the structure of process model is hierarchy, the overall goal of a process model can be divided into some sub-goals that can be assigned to sub-activities in the process model. An activity can promise to achieve multiple goals, while multiple activities can also promise to achieve the same goal. We will prove that it’s decidable whether a process model can achieve some specified goals, so we can try to keep the behavior consistency while evolving a process instance.

Theorem 1. Whether a process model can achieve its goals is decidable.

Proof. Regarding the concurrent objects in process model as a formula of LTL, we can transform them into a Büchi automaton[9] A_O with state spaces that cover all possible status of data in objects. So the goals of process model can be some accepting states in automaton A_O. We will mention that there is a deterministic automaton A_P that equals the pattern constraints of a process model in semantics in the next section, hence the global behavior of the process model can be decided by both automaton A_O and A_P. So the process model can achieve its goals, if and only if the language accepted by both A_O and A_P isn’t empty. Referred to Ref.[9], there is a Büchi automaton A such that $L_\omega(A)=L_\omega(A_O)\cap L_\omega(A_P)$, and the nonemptiness problem for Büchi automaton is decidable. So whether a process model can achieve its goals is decidable. □

Theorem 2. Whether an executing process instance can achieve its goals is decidable.

Proof. Assuming the process instance can be formalized as a deterministic Büchi automaton A, and its current state is s^c. We can construct another Büchi automaton A' for formal analysis, where we set the initial state of new automaton as current state s^c of automaton A. From Theorem 1, whether an executing process instance can achieve its goals is decidable. □

In actual process evolution, the existing result of process enaction should be kept so that the evolved process instance can resume the enaction. So, the current state of process instance should be mapped into the evolved process instance before analyzing it. For mapping current state into evolved process instance, we will analyze the impact of each pattern constraint evolution primitive on the current state of process instance in the following. At first, we denote that “°” is the mark of current state in a pattern constraint, and propose the form of pattern constraint evolution primitives with state mark. Assuming P, P' are pattern constraints, and state marks are located in the pattern constraint P, we can decide the corresponding state of evolved pattern constraint based on the rules below:

- the evolution (1) $P\rightarrow P\vee{}^\circ P'$, (2) $P\rightarrow P\|{}^\circ P'$ add a state mark before additional pattern constraint P',
- the evolution $P\rightarrow(\neg P)\,^\circ$ add a state mark after evolved pattern constraint,
- the evolution $P;P'\rightarrow{}^\circ P'$ add a state mark before pattern constraint P', while delete pattern constraint P,
- other evolution primitives don’t impact on the state mark in pattern constraint P.

Obviously, the state mark can be readily expressed in the formalized Büchi automaton of process model after making some minor changes on the construction steps. So evolved Büchi automaton of process instance can reflect current state.

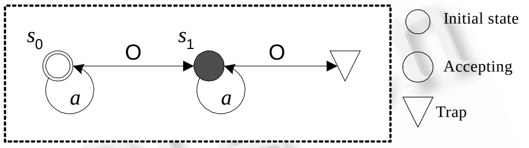
From the proof and the construction step of the Büchi automaton, we can also conclude that the complexity of analyzing process model makes it to be impracticable if constructing the whole automation. On the other hand, some techniques in other Finite State Verification (FSV) approaches, such as SMV[6], SPIN[7], and FLAVERS[8], can be applied to the process model for decreasing complexity. In this paper, we don’t involve the complexity of model analysis, while focusing on how to apply the automatic verification approaches to process domain.

3 Specify Pattern Constraint with Büchi Automaton

Regarding each operation in process model as a character, the execution sequence of operations can be regarded as a word. Intuitively, we can propose an approach to construct Finite State Automaton (FSA) that can and only can accept the words that satisfy a pattern constraint, and we claim the FSA is equivalence of the pattern constraint in semantics. Here a (non-deterministic) finite automaton A is a tuple (Σ^+, S^+, S^0, ρ, F), where $\Sigma^+=\Sigma\cup\{\varnothing\}$, $S^+=S\cup\{\tau\}$, Σ is a finite nonempty alphabet according to operations in the process model and $\varnothing$ is a null operation, S is a finite nonempty set of states, $S^0\subseteq S$ is a nonempty set of *initial* state, $F\subseteq S$ is the set of *accepting* state, and ρ: $S^+\times\Sigma^+\rightarrow 2^{S+}$ is a transition function, i.e. the set of states that A can move into when it's in state s and it reads the symbol a, and τ is a trap state where $\rho(\tau, a) = \tau$ for any symbol a in Σ. Because the execution of a process model may be infinite, the FSA is a Büchi Automaton.

An operation O as atomic pattern constraint is equivalence of the following automaton A, where $S = \{s_0, s_1\}$, $S = \{s_0\}$, $F = \{s_1\}$, and

$$\rho(s,O)=\begin{cases}\{s_1\}, & \text{if } s=s_0\\ \{\tau\}, & \text{if } s=s_1,\end{cases}$$
$$\rho(s,a)=\{s\} \quad \text{if } a\neq O.$$



And the FSA can be represented by the figure above. Obviously, $L_\omega(A)=L_\omega(O)$. Here we use the same notation L_ω for the accepting infinite words for a Büchi Automaton.

Let P, Q be pattern constraint, automaton $A_P=(\Sigma\cup\{\varnothing\}, S_P\cup\{\tau\}, S_P^0, \rho_P, F_P)$ and $A_Q = (\Sigma \cup \{\varnothing\}, S_Q \cup \{\tau\}, S_Q^0, \rho_Q, F_Q)$ is equivalence of P and Q respectively. Without loss of generality, we assume that S_P and S_Q are disjoint. In the following, we construct the equivalent automaton of pattern constraints $P;Q$, $P \parallel Q$, $P\vee Q$, $\neg P$, $[P]$, and $P+$, while omitting P^*.

Proposition 1. ($P\|Q$) Automaton $A = (\Sigma \cup \{\varnothing\}, S, S^0, \rho, F)$ is equivalence of pattern constraint $(P \parallel Q)$, where $S=(S_P\cup\{\tau\})\times(S_Q\cup\{\tau\})\times\{1,2\}$, $S^0=S_P^0\times S_Q^0\times\{1\}$, $F=F_P\times S_Q\times\{1\}$, and $(s',t',j)\in\rho((s,t,i),a)$ if $s'\in\rho_P(s,a)$, $t'\in\rho_Q(t,a)$, and $i=j$, unless $i=1$ and $s\in F_P$, in which case $j=2$, or $i=2$ and $t\in F_Q$, in which case $j=1$.

Proof. Referred to [9], $L_\omega(A) = L_\omega(A_P) \cap L_\omega(A_Q)$, so $L_\omega(A) = L_\omega(P \parallel Q)$.

Proposition 2. ($P \vee Q$) Automaton $A_P \cup A_Q$ is equivalence of pattern constraint $(P \vee Q)$.

Proof: Referred to [9], $L_\omega(A_P \cup A_Q) = L_\omega(A_P) \cup L_\omega(A_Q)$, so $L_\omega(A_P \cup A_Q) = L_\omega(P \vee Q)$.

Proposition 3. ($\neg P$) There is a Büchi automaton A is equivalence of pattern constraint $(\neg P)$.

Proof. There is a Büchi automaton A such that $L_\omega(A) = \Sigma^\omega - L_\omega(A_P)$[9], so $L_\omega(A) = L_\omega(\neg P)$.

Proposition 4. (P;Q) There is a Büchi automaton A is equivalence of pattern constraint (P;Q).

Proof. Considering automaton $A' = (\Sigma \cup \{\varnothing\}, S \cup \{\tau\}, S^0, \rho, F)$, where $S = S_P \cup S_Q$, $S^0 = S_P^0$, $F = F_Q$, and

$$\rho(s,a)=\begin{cases}\rho_P(s,a) \text{ if } s\in S_P \text{ and } a\in\Sigma-\Sigma_Q\cup\Sigma_P\\ \rho_Q(s,a) \text{ if } s\in S_Q \text{ and } a\in\Sigma-\Sigma_P\cup\Sigma_Q\\ \{\tau\} \text{ if } s\in S_P \text{ and } a\in\Sigma_Q-\Sigma_P\\ \qquad \text{or } s\in S_Q \text{ and } a\in\Sigma_P-\Sigma_Q\end{cases}$$
$$\rho(s,\phi)=S_Q^0 \text{ if } s\in F_P$$

obviously, $L_\omega(A')=\{ab|a\in L_\omega((\Sigma-\Sigma_Q\cup\Sigma_P), A_P), b\in L_\omega((\Sigma-\Sigma_P\cup\Sigma_Q),A_Q)\}$, so $L_\omega(A')=L_\omega(P;Q)$.

Proposition 5. (P+) Automaton $A=(\Sigma\cup\{\phi\}, S\cup\{\tau\},S^0,\rho,F)$ is equivalence of pattern constraint (P^+), where $S = S_P$, $S^0=S_P^0$, $F=F_Q$, and

$$\rho(s,a)=\rho_P(s,a) \text{ if } s\in S_P$$
$$\rho(s,\phi)=S_P^0 \text{ if } s\in F_P$$

Proof. Obviously, $L_\omega(A) =\{a^\omega|a\in L_\omega(A_P)\}$, so $L_\omega(A)=L_\omega(P^+)$.

Proposition 6. ([P]) Automaton $A=(\Sigma\cup\{\varnothing\},S\cup\{\tau\},S^0,\rho,F)$ is equivalence of pattern constraint ([P]), where $S= S_P$, $S^0=S_P^0$, $F=F_P$, and

$$\rho(s,a)=\rho_P(s,a) \text{ if } s\in S_P$$
$$\rho(s,\phi)=F_P \text{ if } s\in S_P^0$$

Proof. Obviously, $L_\omega(A) = L_\omega(A_P) \cup \{\varnothing\}$, so $L_\omega(A) = L_\omega([P])$.

Theorem 3. There is a deterministic Büchi automaton A is equivalence of all of the pattern constraints in a process model.

Proof. Because the operation sequence of a process instance should satisfy all of the pattern constraints in the process model, and an alternative of these pattern constraints $P_1,...,P_n$ is the $(P_1\|\ ...\ \|P_n)$, a non-deterministic Büchi automaton A' that is equivalence of the pattern constraints of a process model can be constructed by Proposition 1 to 6. Obviously there is a deterministic Büchi automaton A such that $L(A) = L(A')$[9], so the proposition is correct.

4 Conclusion and Future Work

Process is long-lived, distributed, heterogeneous, and always evolutive to cope with unforeseen environment. We have proposed a flexible process modeling language FLEX and its support system, which consists of the tools of analysis, enaction, monitor, and evolution. In FLEX, the process of evolution is regarded as meta-process, so that FLEX support system can execute the corresponding meta-process model for evolution, and can evolve the evolution model itself reflectively. In this paper, we propose the goal-based approach for process instance evolution, which is a formal approach that focuses on controlling the change of process instance. Our approach for evolution focuses on ensuring the behavior consistency between the evolved process instance and the original instance to avoid ad-hoc change of process model. Because the process instance is evolved, its semantics must be changed to suit environment, so we propose that goals that are specified while modeling the process can be regarded as the criterion of behavior consistency. In other words, if the goals of a process model can be achieved after an evolution step, the evolution is said to keep the behavior consistency of the process model.

Because the possibility of achieving the goals of a process model should be decided in most situations, formal analysis for process instance evolution is necessary. Some existing Finite State Verification (FSV) approaches, such as SMV, SPIN, and FLAVERS, can decide that whether a model is consistent with a property specification. Because most of these mature FSV approaches cope with Büchi automaton for analysis, we specify the semantics of process model and goals to be Büchi automaton in section 4. So the techniques for decreasing time consuming in those existing FSV approaches can also be applied to the analysis of process model. In addition, after evolution of executing process instance, current executing state of each automaton should be located because the process instances should be resumed based on current executing status. We talk about the impact on current state of automaton after modifying pattern constraint with evolution primitives, and give a method to decide whether the evolved process instance can also achieve its goals in successive process enaction.

Current PSSs that support process evolution, such as SPADE, PIE, can provide mechanism for modifying process models or instances, and support the consistency checking among fragments of the process instance, which keeps the static consistency. But there isn’t any process support system that can cope with the behavior consistency between the evolved process instance and the original instance, in order to avoid ad-hoc evolution. Our approach supports to keep the behavior consistency with formal analysis, so users can evolve process instance in reasonable and rigorous way.

Now we are investigating the techniques for decreasing complexity of process analysis with FSV approach because the process model is a special software system, or the effective approach for some special purpose, such as deciding no-deadlock in process instance, or special high-level evolution steps.

References:

[1] 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.

[2] Cunin, P.Y. The PIE project: an introduction. In: Proceedings of the EWSPT7. Kaprun, 2000.

[3] 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.

[4] Chen, Cheng, Shen, Bei-jun, Gu, Yu-qing. A flexible and formalized process modeling language. Journal of Software, 2002,13(8):1374~1381 (in Chinese).

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

[6] McMillan, K.L. Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers, 1993.

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

[8] Cobleigh, J.M., *et al*. FLAVER: a finite state verification technique for software systems. UM-CS-2001-017. ftp://ftp.cs.umass.edu/pub/techrept/techreport/2001/UM-CS-2001-017.ps

[9] Vardi, M.Y. An automata-theoretic approach to linear temporal logic. 2002. http://www.cs.rice.edu/~vardi.

附中文参考文献:

[4] 陈诚,沈备军,顾毓清.一个柔性的形式化过程建模语言.软件学报,2002,13(8):1374~1381.

一种基于目标的过程实例演化方法

陈 诚, 顾毓清

(中国科学院 软件研究所,北京 100080)

摘要: 过程具有长生命周期、分布式结构、异构等特性.为适应不可预料的环境变化,过程还需要不断地演化.提出一种过程实例的演化方法以控制对过程实例的改变.它可以检查并保证演化前后的过程实例具备"基于目标的行为一致性",从而避免用户对过程的盲目修改.目标作为行为一致的标准,需要分析演化后的过程实例来保证它可以完成目标.通过用 Büchi 自动机描述过程实例和其目标的语义,一个过程实例是否能够完成其目标是可判定的.

关键词: 过程实例;过程模型;演化;基于目标

中图法分类号: TP311 文献标识码: A