Analysis and Enactment of A Distributed Software Process Model : The AttNet Model

Woo Jin Lee†, In Sang Chung‡, and Yong Rae Kwon†
†Department of Computer Science, Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon 305-701, Korea
E-mail: {woojin,yrkwon}@salmosa.kaist.ac.kr
‡Department of Computer Science, Hallym Univ.
1, Okchon-dong, ChunChon, KangWon-do 200-702, Korea
E-mail: ischung@sun.hallym.ac.kr

Abstract

Process modeling has increasingly attracted in the software engineering community. In the process modeling, the enactment of software process model is a dominant theme. In order to correctly and smoothly enacting the software process model, it is necessary to early detect the inconsistencies contained in the software process model before the software process model is instantiated to actual enactment engine. In this paper, we provide an analysis framework and an enactment mechanism for the AttNet model which is based on Petri-nets and supports the distributed software processes.

1 Introduction

Process modeling has increasingly attracted attention in the software engineering community. One dominant theme in process modeling is the notion of a process-centered environment, which operates as an enactment engine for a specific process modeling language[11]. For correctly and smoothly enacting a software process model, it is necessary to early detect its inconsistencies which are caused by modeling errors, before the software process model is instantiated to actual enactment engines. Analysis of a software process model is dependent to its software process modeling language since it is carried out considering the characteristics of the target software process model language.

Many software process model languages are based on notational paradigms originally devised for other purposes. APPL/A[6] extends Ada, Funsoft nets[3] is based on Petri-nets, Marvel[5] is described by rules, HFSP[4] is based on functional language, *etc.* Recently, there have been the efforts to dealing with the support of distributed software processes as software projects become more larger and more complex. In distributed software development environment, different teams which may be located at different sites, have different views on the software processes. According to roles in a team or life cycle phase, a particular subset of software processes may be emphasized. Therefore, the knowledge about software development in such projects should be distributed over the particular teams. In describing this distributed software processes, it is important to independently divide the software processes into geographically distributed teams and to properly integrate each tasks into entire model.

There are some approaches to describe this distributed software processes. ASL[11] is extended the Marvel by representing global control flow and synchronization over local constraints which are described by rules. SLANG[8] is based on Petri-nets. It provides a language construct, *activity*, which represents a unit of execution. As a similar approach, there is the AttNet[1] which is also based on Petri-nets. But it is different with the SLANG approach in the basic idea of language construct, *activity object* and the communication mechanism among them. The detail comparisons will be given in *Related Works*. The AttNet is an adaptation of OPNets[2] which introduced the concept of objects into Petri-nets in order to model the real-time system by Petri nets objects. In AttNet, each software process is described independently using activity object and communications among them are established by message-passing.

Although there have been many software process model languages, only a few of them support analysis frameworks for checking their consistencies[3, 9]. In

0-8186-6960-8/94 $04.00 © 1994 IEEE

this paper, we provide a framework for analysis and an enactment mechanism for a distributed software process model, the AttNet model. Since the AttNet model is hierarchically constructed by activity objects which are divided into visible external structures and invisible internal structures from outside, analysis of the AttNet model is performed along the hierarchy of activity objects by bottom-up approach. At first, the activity objects in the bottom-level of the hierarchy are analyzed using analysis methods. For analyzing a higher-level activity object which contains other activity objects in its internal structure, internal activity objects are abstracted about their reachability properties and are represented in Petri-nets forms. In the abstracted net, the analysis is carried out like a bottom-level activity object.

In enactment, the AttNet model is considered as a pool of activity objects regardless of calling relationships between activity objects. When each activity object is invoked, it is copied from the pool of activity object and instantiated to an enactment process. The enactment process executes its own actions and passes messages between internal activity objects as a message handler. Like the SLANG[8] approach for the computational reflection, the AttNet provides reflection and evolution of process model by using a special activity object, *modify-model*, which gets an activity object from the pool and modifies it and puts it into the pool again. The *modify-model* activity object is similar to the *assert* and *retract* rules in Prolog.

The rest of the paper is organized as follows. Section 2 introduces the AttNet and gives its formal definition and shows an example described by the AttNet. Section 3 explains a hierarchical analysis framework and provides the abstraction algorithm of activity objects. Section 4 provides an enactment mechanism of the AttNet model. Section 5 surveys related works. And conclusions are given in Section 6.

2 The AttNet model

The AttNet was proposed for supporting distributed software processes with a language construct, *activity object*, which represents a unit of task. As an activity object can be described its task with other sub-activity objects, the AttNet model is hierarchically constructed by activity objects. Activity objects can be performed concurrently at different sites and communicate with each other by message-passing mechanism.

2.1 Activity object

Since the internal details of each activity object performed by each team should be hidden and only external interface should be visible from outside in the distributed software environment, activity objects in the Attnet separate internal and external structure clearly. The internal structure in activity object describes what an activity object performs by Petri-nets notations. And the external structure represents interfaces among other activity objects with input and output message queues. In Figure 1, the external

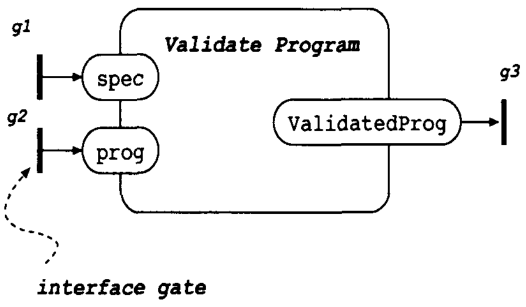


Figure 1: A representation of an activity object

structure of activity object, *Validate Program*, have two input message queues and one output message queue, but its internal structure is not visible. Arrival of tokens at some of input message queues is regarded as the request of services from other activity objects. With input data, it performs its own functionality by enacting a fragment of Petri-nets. Output message queues contain messages which may require the communications with other related activity objects, after performing its services. There are two types of activity objects : *primitive* activity object which has no other activity objects in its internal structure and *composite* activity object which contains other activity objects.

2.2 Communication between activity objects

Activity objects communicate with each other through input and output message queues connected to the interface gates. The interface gate is a sort of transition and passes messages from one output queue to one input queue by firing it. Each interface gate can be considered as an envelope of an executable procedure consisting of two major concurrent processes, *Sender* and *Receiver*. Conceptually , *Sender* is a process which transmits messages from the output message queue of one activity object to *Receiver* and *Receiver* is a process which receives messages from *Sender* and places them to the input message queue of other activity object. The interface gate is appeared in Figure 1.

2.3 Definition of the AttNet

The activity object in the AttNet model is a modeling entity which may contain other activity objects in its internal structure. Thus, the structure of the AttNet model is similar to the tree structure, where each activity object corresponds to node in a tree and *child* activity object is contained in its *parent* activity object. The AttNet model is recursively defined with an activity object, which is similar to defining the tree structure. We choose Predicate/Transition Net[17] as a Petri-Net notation for describing the internal structure of activity objects. Each place in Pr/T net has a strongly typed token which represents data type to be performed.

Definition 2.1 *The AttNet model*

Activity object $AN = (O, T, F; I_Q, O_Q; A, M^0)$

1. $O = S \cup AN'$ *,where*
   AN' *: set of internal activity objects,*
   $S = S_{state} \cup S_{tool} \cup S_{final-product}$
2. $T = T_{act} \cup T_{gate}$*, where*
   T_{act} *: set of transitions,*
   T_{gate} *: set of interface gates*
3. I_Q *= set of input queues*
4. O_Q *= set of output queues*
5. F *= set of arc,*
   $F \subseteq ((O_S \cup S) \times T) \cup (T \times (I_S \cup S))$*, where*
   $I_S = I'_Q$*(set of input queues in* AN'*)* $\cup O_Q$,
   $O_S = O'_Q$*(set of output queues in* AN'*)* $\cup I_Q$
6. A *= annotations for place, transition and arc*
7. M_0 *= initial marking*

2.4 An Example

The example refers to one representative subprocess of *Develop Change and Test Unit*[12] in ISPW6, that of making the design and code changes necessitated by requirements change. This subprocess is represented by four activity objects which may be performed concurrently. Figure 2 shows the interfaces among four activity objects. Internal structures of each activity object are not visible. Each activity object which is contained in *Develop Change and Testing* is further defined with another activity object on the lower level of the hierarchy in details. Each activity object is described as follows :

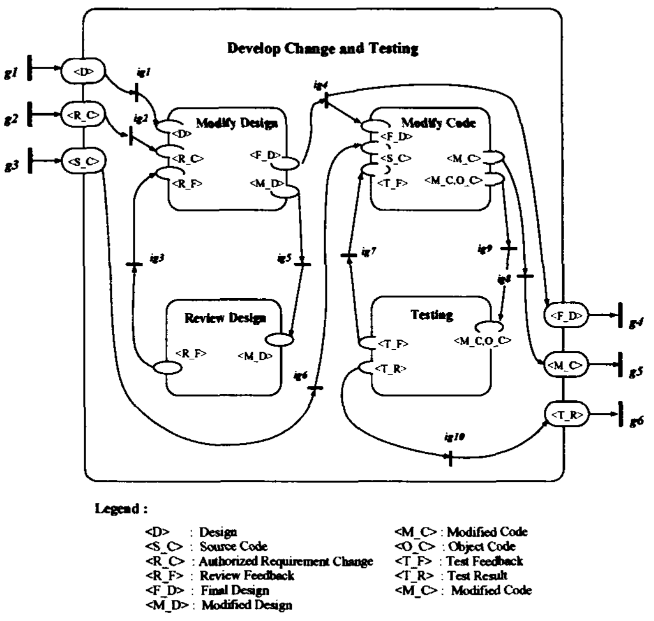


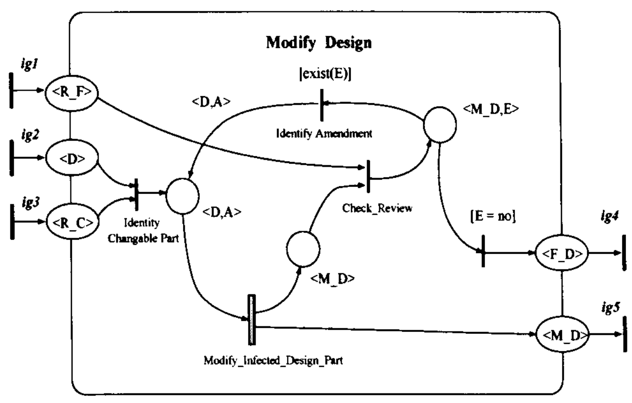
Figure 2: Example: Develop Change and Testing

- *Modify design* activity object modifies the current design with regard to requirements change. This activity object communicates with *review design* activity object for checking the consistency of the modified design.
- *Review design* activity object reviews the modified design for keeping the design consistent.
- *Modify code* activity object modifies the infected part of codes with regard to modification of design. For testing the modified code, this activity object interacts with *testing* activity object.
- *Testing* activity object is to test the modified codes. Testing activity object may be divided into several activity objects : *Modify Test Plans, Modify Unit Test Package, and Test Unit.*

Figure 3 shows the internal structure of activity object *modify design*. In Figure 3, the thick transition such as *modify-infected-design-part* means that it will be further described in details.

3 Analysis of The AttNet Model

Analysis of software process model aims at detecting errors and inconsistencies in software process models before models are used for governing software



Legend :

<R_F>	: Design Review Feedback	<M_D>	: Modified Design
<D>	: Current Design	<E>	: Design Error
<R_C>	: Requirement Changes	<F_D>	: Final Design
<F_C>	: Feedback Changes	<A>	: Amendment of Design

Figure 3: Modify Design process

processes[3]. Since the AttNet model consists of activity objects which represents distributed software processes, its analysis should include local analysis of each activity object and checking for inconsistencies in the communications among activity objects. Analysis of the AttNet model is carried out along the hierarchy of activity objects by bottom-up approach. Figure 4 shows the overall procedure of the analysis for the AttNet model. At first, bottom-level activity objects are analyzed and then they are abstracted about its reachability, which are used for analyzing activity objects in higher level. In order to analyze a composite activity object, internal activity objects of the activity object are replaced into their abstract forms and then, in the replaced net, the analysis of the activity object is to be performed. Considering hierarchical analysis of the AttNet model, the analysis procedure is divided into two main sub-procedures. One is to independently analyze one activity object regardless of other related activity objects. The other is to abstract each activity object about the reachability property which we want to analyze for checking the inconsistencies in communications among activity objects in the composite activity objects.

```
present level = bottom
while (there is no activity object)
        step 1 : for each activity object in present level
                    replace each internal activity object
                    into its abstract form
        step 2 : analyzes each activity object
        step 3 : abstracts each activity object about its
                    reachability
        step 4 : moves to upper level
```

Figure 4: Analysis of The AttNet Model

3.1 Analysis of an activity object

Analysis of each activity object is performed in two sides : static properties and dynamic behaviors. In analysis for static properties, structural properties which were defined in [3], such as source channel, useless object type, sink channel, unprocessable object type, sink agency and so on are treated by checking connectivities between places and transitions. These syntactic errors are mainly caused by mistakes of process model designers. And they are easily checked during writing model since they are visible in Petri-Nets graphical notations. But, for checking dynamic behaviors, we have to construct a reachability tree of each activity object. In order to simplify construction of a reachability tree in PrT net, we don't consider the transition annotation(transition selector). The transition selector is used to choose one transition from some firable transitions which have the same preset. Since transition selectors are used for making a net deterministic, it only increases nondeterminism to get rid of transition selectors in net. Assuming that all input data are available at beginning, the reachability tree of activity objects can be constructed. Using reachability tree, we checks following properties:

- **liveness :** each action in an activity object should be firable at least once. And there is no dead sequence in a reachability tree.
- **reversibility :** a Petri net(N, M_0) is said to be reversible if, for each marking M in $R(M_0)$, M_0 is reachable from M[16].
  In a reversible net, it always get back to the initial marking or state.

For checking liveness, we search for the dead state node: that is, the state which any transition cannot be firable and transitions which are not appeared on arcs in reachability tree. In the AttNet model, a reversible activity object means it will always do the same actions given the same input data. For satisfying the reversibility, each terminal node can be back to the root node in the reachability tree.

step 1 : constructs a reachability tree
step 2 : obtains ordering relationships from a reachability tree
step 3 : constructs Extended Hasse Diagram
step 4 : transforms Extended Hasse Diagram into an IEN

Figure 5: Abstracting an activity object into an IEN

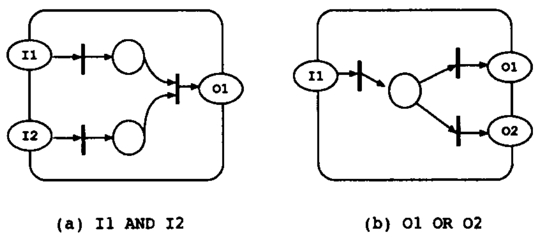


Figure 6: AND, OR relations in ordering relations

3.2 Abstraction of activity objects

An activity object consists of a visible external structure and an invisible internal structure from outside. Since the internal structure of an activity object cannot be seen from outside, there should be a mechanism which can represent its characteristics using its visible external structure, for analyzing composite activity objects. As a property for characterizing the behavior of an activity object, we use the ordering relationships of its input and output queues, which are used for constructing reachability tree in composite activity objects. For representing these ordering relationships with Petri-net notations, we generalized the Interface Equivalent Net(IEN) concept which was proposed in order to analyze interactions among sequential objects in OPNet[2]. An IEN is a Petri-net, where transitions represent input, output queues and places are used for achieving ordering relationships between the external structures. That is, an IEN is a abstract representation of an activity object about its reachability property from outside views. Figure 5 shows the overall procedure for abstracting an activity object into an IEN form. The ordering relationships are extracted from its reachability tree and then they are represented *Extended Hasse Diagram(EHD)* which contains AND, OR relations on the branch over Hasse Diagram and finally, it is transformed into IEN.

3.2.1 Extracting ordering relationships

A reachability tree contains all possible ordering sequences of firable actions. From the reachability tree, we can get ordering relations of all pairs of input and output queues. These ordering relations satisfies the condition of a partial ordering relation ,eliminating two special cases which are appeared in Figure 6. In Figure 6 (a), joining and branching at a transition cause all related queues to be executed in arbitrary order, which we define as a *AND relation.* In Figure 6(b), joining and branching at a place cause one of related queues to be executed , which we define as a *OR relation.*

Definition 3.1 *Extended Hasse Diagram*
Hasse Diagram is used for graphically representing the partial ordering relations. Extended Hasse Diagram is a Hasse Diagram which contains AND, OR relations on each branching and joining.
An AND relation among places means that all related places have to be passed. An OR relation among places means that one of related places may be passed.

The ordering relationships of the external structure can be described with Extended Hasse Diagram.

3.2.2 Transforming into an IEN

An EHD is transformed into an IEN by transformation rules shown in Figure 7. Input and output queues in EHD are transformed into transitions in IEN. For reserving the ordering relations, places are inserted between transitions in IEN. In Figure 7, a sequence of two queues is represented as a sequence of place and a transition. And the AND or OR relations in the EHD are transformed the branching and the joining on transitions or places, respectively. Transitions in the IEN are used for synchronizing with other activity objects : input queue synchronizes with incoming arc into that transition, output queue with outgoing arc from that transition.

Figure 8(a) shows *Develop Change and Testing* activity object whose internal activity objects are replaced its IEN. In Figure 8(a), SubNet1 is the IEN for Modify Design, SubNet2 for Review Design, SubNet3 for Modify Code, and SubNet4 for Testing. In Figure 3, we can easily extract the ordering relationships of external gates(ig1 - ig5), (ig2, ig3 ; ig5; ig1; ig4). That is, in Figure 3, *Design* and *Requirement Change* are used for generating *Modified Design* and then *Review Feedback* from activity object, *Review Design,* is used for producing *Final Design.* These ordering relationships which are represented in EHD are trans-

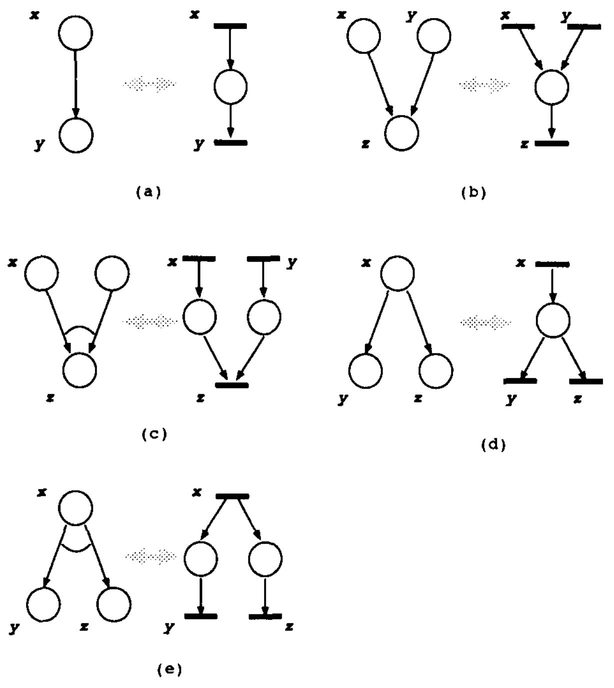
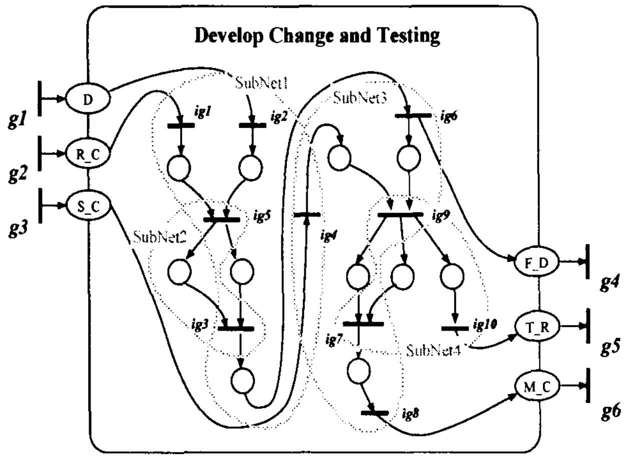


Figure 7: Rules for transforming from EHD to IEN

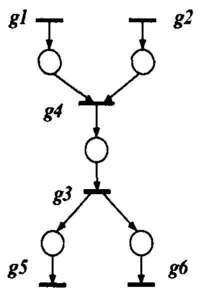
formed into an IEN, SubNet1, by applying transformation rules. Each SubNet is merged into one net by unifying the same interface gates in each SubNet. That is, input and output queues which are the same data type are unified into one transition in a merged net. We call this merged net as *abstract object*. Figure 8(b) is the IEN which is obtained by applying the abstraction procedure to the abstract object of Figure 8(a). In Figure 8(a), we can get the ordering relationships among external interface gates, (g1, g2; g4; g3; g5, g6). Analysis of abstract object is carried out by the same analysis method which was described in section 3.1.

4 Enactment of The AttNet Model

The AttNet model can be considered as a pool of activity objects although they are logically connected in a hierarchical structure. Also, each activity object has its own life-cycle : after it is invoked at a stimulus, it performs its own function and finally dies. In enacting the AttNet model, activity objects are instantiated from the pool of activity objects, like rules in rule-based system. Figure 9 shows the overall structure of enacting the AttNet model. Since each activity object is invoked by its parent activity object, running enact-



(a) After abstracting internal activity objects



(b) The IEN of (a)

Figure 8: An example for IEN

ment processes construct a tree structure,*enactment process tree*, by calling relations. In Figure 9, an enactment process B is invoked by its parent process A and is to be a child of parent process. Each process on enactment process tree has two roles : one is to invoke sub-activity objects and enact actions in its internal structure; the other is to pass messages among child enactment processes as an message handler. In Figure 9, an enactment process, A, plays a role as an message handler of enactment processes B,C, and D.

There is one special activity object which modifies the AttNet model itself. In Figure 9, enactment process F modifies activity objects in the pool. Using this activity object, we provide the reflection and evolution properties of software process model. Like SLANG[8] approach for providing computational reflection, we treat the AttNet model as data which can be represented in tokens. Figure 10 shows the spe-

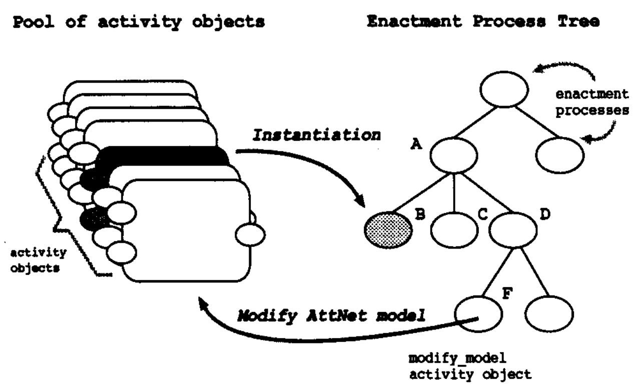


Figure 9: The overall structure of enacting the AttNet model

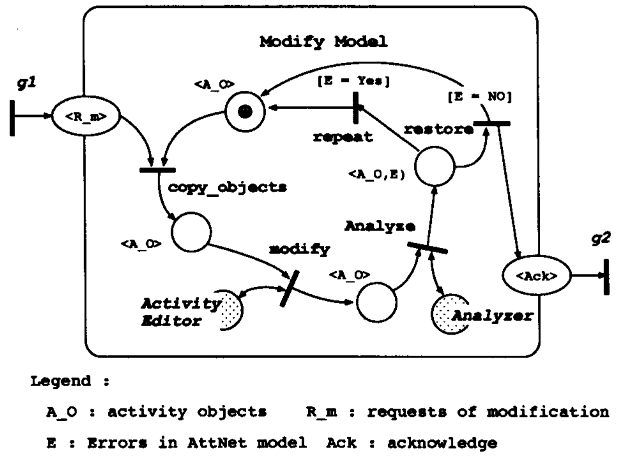


Figure 10: activity object *modify model*

cial activity object,*modify model.* After it receives the request of modification as an input, it copies activity objects which we want to modify from the pool of activity objects and then it modifies activity objects. The modified AttNet model have to be checked its consistency by analysis methods before it is put at the pool again. Like *assert* and *retract* rules in Prolog, the modify-model is used at changing the activity objects of the pool.

4.1 Enactment Process

Each activity object is instantiated into an enactment process which consists of four important parts. They are described as followings :

- *local status* keeps the current state of the enactment process. It is updated by other three parts.
- *execution engine* selects firable actions by referring to the local status. Then it enacts one of firable actions. And it invokes activity objects which include in its internal structure.
- *message handler* passes messages among child-enactment processes.
- *exception handler* treats the issues involved in on-the-fly modification of process(that is, rollbacking, undoing, passing steps and so on) by updating the local status in order to keep enactment processes consistent. Also, this part has the full responsibility for whether to terminate this enactment process or not.

An enactment process is invoked by an execution engine of parent's enactment process. Its termination is decided by the exception handler by checking the local status. When the present state of local status is the same of the initial state, the exception handler terminates its enactment process and informs its parents process of termination.

4.2 Evolution of process model

Activities for evolving software process model are appeared in root activity object of the AttNet model. Figure 11 shows the mechanism for supporting evolution of process model. Main software processes and an evolution process are concurrently executed. The evolution process is considered as a monitoring process for main software processes. That is, it is monitoring the request of changing model and, if any request, it invokes the modify-model activity object and finally returns to the initial monitoring state. The heart of the evolution process is the modify-model activity object. The evolution process will be terminated after the final product is delivered.

5 Related Works

There have been many process modeling languages which are based on notational paradigms originally devised for other purposes[10]. But, a few process modeling languages support the distributed software processes. In describing the distributed software processes, it is important to clearly separate tasks of each team and to properly integrate them into entire process model for the convenience of describing and analyzing software processes.

As distributed software process modeling languages, there is a ASL[11]. ASL provides bi-level

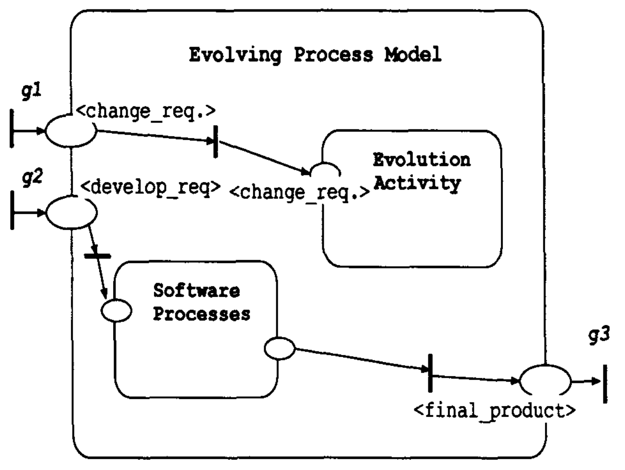


Figure 11: activity object *modify model*

formalisms suitable for expressing and enacting large-scale software processes. The global formalism concentrates on the overall control flow and synchronization among processes. And local formalism is used for expressing constraints and policies on individual tools and data. Since ASL combines two different formalisms for describing distributed software processes, it is difficult to check the consistency of software process model. But, the AttNet can support global and local formalisms using a single language construct, activity object.

SLANG[8] is based on Petri-nets for describing distributed software processes. It is very similar to the AttNet approach in hierarchically constructing process model with activities and enacting the model by instantiating the activities. But, they have differences in their basic language constructs: SLANG's activity vs. AttNet's activity object. Followings are basic differences of two approaches.

- In the AttNet, activity objects are based on the object concept with the message passing mechanism for communicating between activity objects. But, in SLANG, activities communicate and are synchronized via global data, interface places. That is, while SLANG is tightly coupled, the AttNet is loosely coupled.

- Since SLANG communicate via global data, there are some difficulty in analyzing each activity independently. Therefore, entire model is considered at the same time rather than one by one in analysis. In AttNet, the activity object may be considered as an independent object. So, it is possible to analyze each activity object and then merge the analyzed results for checking global consistencies.

In the view of the checking the consistency of software process model, there are a few process model languages. DesignNet[9] is based on Petri-nets and AND-OR graphs. AND-OR graphs are used to decompose goals into subgoals and Petri-nets describes process model skeletons. In DesignNet, some properties of project are analyzed: connected, plan complete, plan consistent, and well-executed. Funsoft nets[3] is also based on high-level Petri-nets, Predicate/Transition net. With transitions refinement, Funsoft nets is constructed hierarchically. In Funsoft nets, there are a variety of analysis methods for static and dynamic properties of a software process model. Although these two nets provide some analysis methods by adapting a variety of Petri-nets analysis methods, there are no language constructs for describing and analyzing distributed software processes.

6 Conclusions

In this paper, we presented an analysis framework and enactment mechanism for a distributed software process model, the AttNet model. Considering the hierarchical structure of the AttNet model, the hierarchical analysis is performed by using the abstraction of activity objects. The abstraction algorithm extracts the reachability property from the internal structures of activity objects and represents it in abstract net using the same Petri-nets form. In executing the AttNet model, activity objects are considered as independent items in the pool. After being copied from the pool, activity objects are instantiated to enactment processes, which are concurrently executable units at different sites. For providing the concepts of reflection and evolution of process model, a special activity object, modify-model, which modifies activity objects is proposed.

Since this work represents at the minimum of analysis framework and enactment mechanism for the AttNet model, the further researches on describing and analyzing resources and data types which are defined as tokens are to be done.

References

[1] I.S. Chung and Y.R. Kwon, "The AttNet Model: A Petri Net Based Language for Modeling Dis-

tributed Software Processes", *Proc. InfoScience '93*, pp. 121-128, Oct 1993.

[2] Y.K. Lee and S.J. Park, "OPNets : An Object-oriented High-level Petri Net for Real-time System Modeling", *Journal of Systems and Software*, Vol. 20, pp. 69-86, Jan 1993.

[3] V. Gruhn, *Validation and Verification of Software Process Models*, University Dortmund, 1991.

[4] T. Katayama, "A Hierarchical and Functional Software Process Description and its Enaction", *Proc. 9th Int'l Conf. on Software Eng.*, pp. 343-352, 1989.

[5] G.E. Kaiser, P.H. Feiler and S.S. Popovich, "Intelligent Assistance for Software Development and Maintenance", *IEEE Software*, pp. 40-49, May 1988.

[6] S.M. Sutton, L.J. Osterweil and D. Heimbiger, "Language Constructs for Managing Change in Process-Centered Environments", *Proc. 4th ACM SIGSOFT Symposium on Software Development Environment*, pp. 206-217, 1990.

[7] S. Bandinelli and A. Fuggetta, "Computational Reflection in Software Process Modeling : the SLANG Approach", *Proc. 15th Int'l Conf. on Software Eng.*, pp. 144-154, 1993.

[8] S. Bandinelli, A. Fuggetta and C. Ghezzi, "Software Process Model Evolution in the SPADE Environment", *IEEE Trans. on Software Engineering*, Vol. 19, No. 12, Dec. 1993.

[9] L.C. Liu and E. Horowitz, "A Formal Model for Software Project Management", *IEEE Trans. on Software Engineering*, Vol. 15, No. 10, pp. 1280-1293, May 1989.

[10] B. Curtis, M.I. Kellner and J. Over, "Process Modeling", *Comm. of the ACM*, Vol. 35, No. 9, pp. 75-90, Sep 1992.

[11] G.E. Kaiser, S.S. Popovich and I.Z. Ben-Shaul, "A Bi-Level Language for Software Process Modeling", *Proc. 15th Int'l Conf. on Software Eng.*, pp. 132-143, May 1993.

[12] M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil and M.H. Penedo, "ISPW-6 Software Process Example", *Proc. 1st Int'l Conf. on the Software Process : Manufacturing Complex Systems*, pp. 176-186, Oct 1991.

[13] D. Heimbiger and M.I. Kellner, "Software Process Example for ISPW-7", *available in /pub/cs/techreports/ISPW7/ispw7.ex.ps.Z by anonymous ftp ftp.cs.colorado.edu*, Aug 1991.

[14] B. Peuschel, W. Schafer and S. Wolf, "A Knowledge-Based Software Development Environment Supporting Cooperative Work", *International Journal of Software Engineering and Knowledge Engineering(SEKE)*, Vol. 2, No. 1, pp. 79-106, 1992.

[15] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1986.

[16] T. Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-580, Apr 1989.

[17] H.J. Genrich, "Petri Nets: Applications and Relationships to other Models of Concurrency", *Predicate/Transition Nets*, edited by W. Brauer, W. Reisig, G. Rozenberg, Springer-Verlag, pp. 207-247, 1987.