Incremental Scenario Modeling Using Hierarchical Object-Oriented Petri Net

Jang-Eui Hong and Doo-Hwan Bae
Devision of Computer Science
Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon 305-701, Korea
Tel: +82-42-869-3539, Fax: +82-42-869-3510
{jehong,bae}@salmosa.kaist.ac.kr

Abstract

Scenario-based modeling is emerging as a key technology in software development because scenarios are easy to describe and understand, naturally describing the real world, and highly used in several development phases. Scenario-Based Software Modeling(SBSM) represents scenarios with a formal method, and then integrates each scenario into a global one excluding redundant, incomplete, and inconsistent elements. An issue which is not yet proposed a clear solution in SBSM is the handling of different abstraction levels. In the existing proposals, this problem is hidden with an implicit assumption that all such scenarios have the same abstraction levels, which we believe to be unrealistic. In our research, we propose a Hierarchical Object-Oriented Petri Net(HOONet) as a method to specify the scenarios, and also suggest a method to integrate scenarios including different abstraction levels as well as redundancy, incompleteness, and inconsistency. Coordinating abstraction levels of scenarios can be achieved by considering scenario models as state-based transition models and verifying the transition models in respect to behavioral equivalence. In order to show the effectiveness of our suggestion, overall processes of SBSM using HOONet are explained with a practical example.

Keywords: *scenario modeling, hierarchical abstraction, formal methods, object-oriented Petri net, incremental modeling*

1 Introduction

Software modeling based on scenarios is a widely used method in the development of industrial applications because the method can naturally represent user requirements and drive user's attending in requirements analysis work. Scenarios are the description of requirements representing the interaction between system and actors, and they provide the multiple views in the several aspects of the system[7, 13].

The main problems in the scenario-based software modeling process are incompleteness caused by insufficient scenarios for the whole system behaviors, and inconsistency caused by redundant and/or overlapping scenarios for the same behavior. To cover these problems, the scenarios described by natural language are transformed into formal representations, and then the transformed scenarios are integrated into a global scenario. In this process, the incompleteness and inconsistency problems can

be avoided by the verification techniques of formal methods[16, 23, 27].

However, scenarios can represent different views of a system and can be described with different abstractions because they are elicited from multiple views of various users. These scenarios should be integrated into a global scenario after coordinating the different abstraction levels of each scenario to an equivalent level. However, existing researches on scenario integration have the strict assumption that each scenario has the same abstraction level. This assumption is not realistic. Rolland[24] and Desharnais[9] specifically pointed out that the abstraction problem must be solved in scenario-based software modeling.

In this paper, we suggest a modeling method of system requirements based on scenarios, which introduces an object-oriented formal technique to coordinate the different abstraction levels of scenarios. Our method, HOONet(Hierarchical Object-Oriented Petri Net) adopts object-oriented concepts to support the hierarchical abstractions and Petri net formalism to formally represent scenarios. The HOONet method can model the different abstraction levels of system states and behaviors, and is able to incrementally perform the modeling and analyzing when the abstraction is not fully refined yet. Because HOONet models can be analyzed by the incremental and compositional approach, this approach allows us to reduce the state explosion problem which can occur in the analysis of integrated scenario models.

This paper is organized as follows: In Section 2, we briefly review some of the earlier proposals made for scenario-based modeling and identify their shortcomings. In Section 3, we first give some consideration issues for scenario modeling, and justify a need for defining HOONet by explaining what is required to accomplish our goal. We provide an informal and intuitive introduction to HOONet as well as formal definitions. Section 4 describes a systematic procedure for modeling a system using HOONet, and explains the procedure through modeling an example system represented with abstract information. Integrating scenarios and coordinating their abstraction levels are explained in Section 5. In Section 6, analysis techniques for ensuring the correctness of structural and behavioral properties in HOONet models are discussed. We provide the algorithms to generate a reachability graph of HOONet models. Section 7 concludes our paper and suggests further research issues. In Appendix A and B, the detail descriptions of HOONet formalism are given.

2 Related Work

Recently, the existing researches on scenario-based software engineering have focused on scenario integration to represent the global behavior of a system[10, 16, 17, 28]. Our research, on the other hand, focuses on the handling of the different abstraction levels in the scenario integration. Such focus comes from the observation of the developer's interests in the industrial application development using

scenarios.

Hsia et al.[11] suggested the procedures for scenario analysis, which use scenario trees and conceptual state machines to represent and analyze the scenarios. They study the techniques for scenario formalization, generation, and verification, but do not mention the issues of abstraction and integration of scenarios.

In Glinz's research[10], each scenario is transformed into statechart formalism, and is integrated into a global behavior. To integrate each formally represented scenario, he suggests the composition rule for statechart templates which represent the relationship among the scenarios such as sequence, alternative, iteration, and concurrency. Abstraction of a specific behavior or an actor is represented with a simplified statechart. This statechart is represented by only its name, omitting its internal behaviors. Thus, it cannot represent the interaction among its components and the state changes of the abstract component. Moreover, we believe that the notion of abstraction in Glinz's research is not the representation of the high-level information elicited from user requirements, but only the omitted representation of statecharts to reduce the complexity of the drawn models. After the completion of all requirements with statecharts, each model can be abstracted(strictly speaking, can be simplified) with a simplified statechart.

Regnell suggested the UORE(Usage Oriented Requirement Engineering) method[23], which is an extension of use-case based analysis and design method. In this method, the use-case model is represented with message sequence charts. There are two considerations for abstraction in the UORE method. One is the abstraction of the interface part between users and systems. This postpones the design decision of the interface parts which consist of menu buttons, peripheral devices, and so on. The other is the abstraction for model itself. This simplifies the complex representation of models and improves the understandability and readability of the drawn models. However, these two cases do not support the real notion of abstraction. For the first case, it is not the abstraction of system behaviors and states. The abstraction which is related to the hardware interface is one of the natural design techniques in common software development. The second case is the same as the abstract representation in Glinz's research.

The integrating method of scenarios represented with timed automata is suggested by Some et al.[27, 28]. This method defines the interactions within a scenario in the first step, and then, the stimuli and reactions for each interaction are identified. Scenario integration is processed by the evaluation of the conditions for the stimuli and their reactions. The inconsistency and incompleteness of the scenarios are detected by the evaluation of the conditions. The integrated scenario is represented by a global automata. This method does not consider the notion of abstraction because the condition of each interaction has specific timing constraints.

Desharnais et al.[9] integrated state-based sequential scenarios. This method represents scenarios with a formal relation-based definition and integrates the scenarios into a more global view of user-system interaction. Scenarios are represented by means of relations, and their integration is defined by means of relational operations. However, the restriction of this method is that the scenarios are sequential cases for only one user. Thus, concurrency of the system cannot be expressed in the scenario. In integrating sequential scenarios, the inconsistency and incompleteness problems are considered, but the problem on the different abstraction levels of each scenario remains in further works.

Other research includes the scenarios integration by finite state automata[17], the specification and composition of scenarios represented with high-level Petri net[16], the representation of interrelation of scenarios by message sequence charts[3], and so on[1, 4, 19]. The researches mentioned above are interested in the scenario integration considering the inconsistency and incompleteness problems, but do not consider the notion of abstraction in scenario representation. Therefore, we propose a scenario modeling and analysis method, based on object-oriented Petri net, which provides the hierarchical representation of scenarios according to their abstraction levels, and also provides the incremental integration of scenarios satisfying the requirements of consistency and completeness.

3 Hierarchical Object-Oriented Petri Net

3.1 Motivations

The purpose of designing HOONet is to provide a modeling language which combines object-oriented concepts and strict formalism[12]. There are some issues which should be considered in representing scenarios with a given modeling language. HOONet supports the following issues which should be considered in representing scenarios:

- Modularization: Scenarios are decomposed into entities and their interactions to analyze the behaviors of a system or to represent the scenarios with formal description techniques. These decomposed entities should be modularized for low coupling among them and for the handling of redundant or overlapped scenarios.
- Abstraction: The entities should be able to represent abstract information at high-level of decomposition structure. This abstraction should be refined at lower-level. In scenario modeling, such abstraction and refinement techniques should be supported, and straight modeling of abstractly represented entities in decomposition structure should be possible without the loss of information.
- Specialization: Scenario decomposition can produce the hierarchical structure of the system entities. The low-level entities specialize the high-level entities. By this specialization, the

low-level entities can inherit all or some properties of the high-level ones.

- Interaction: The interactions between actors and (sub)systems are represented by communication mechanisms. These mechanisms should be able to support the interaction methods using message passing and global variable accessing.

We strongly believe that Petri nets are well-suited to represent the above issues by their well-defined semantics, their ability to model concurrent and asynchronous system behavior, and the variety of mature analysis techniques they offer(e.g., reachability, deadlock, invariant, etc). Unfortunately, both classical Petri nets, known as place/transition nets(P/T nets)[22], and high-level Petri nets such as colored Petri nets(CPN)[14] are inadequate for our purpose. First, they do not provide adequate language constructs supporting modular and incremental modeling. The model based on P/T nets is not only difficult to understand, but also quite sensitive to changes since the P/T nets model the behaviors with the bottom-level information of a system. second, it is difficult to perform selective behavioral simulation tailed to the user's interests. Third, the notion of abstraction of both data and operation is not supported.

CPN, a high-level Petri net formalism, defining the tokens which can be arbitrary color(type) and complexity, provides advanced features such as substitution transition and fusion places to support modularity. Substitution transition utilizing the port concept is a notational convenience designed to allow modular representation of large and complex models. However, all "atomic" transitions model events at the same level of abstraction, although they may appear on different and hierarchically organized pages. Fusion places are used to avoid the clustering of too many input and output arcs. Fusion places appearing on different pages are considered the same, and the firing semantics of CPN are unaffected, while clearly the understandability of CPN models and fusion places alone from the viewpoint of formalizing scenarios are insufficient to fulfill the above issues.

Another trend in Petri nets research is to introduce object-oriented concepts. Examples include OBJSA[2], COOPN/2[5], LOOPN++[15] and G-Nets[8, 20, 21, 26]. Although these Petri nets extensions support the object-oriented concepts to handle the issues above, they are not sufficient to support the incremental scenario modeling. The modeling of large and complex systems should be performed with hierarchical decomposition of scenarios along with the abstracting and specializing of them. However, we believe that the existing proposals on object-oriented Petri nets do not provide the notions which are abstracting both data and operations as well as overriding properties in aspects of specialization[12]. In [12], we survey and compare some object Petri nets - such as OBJSA, COOPN/2, LOOPN++, OPNets, and G-Nets - in aspects of supporting the concepts of object-orientation.

Critical examination of various proposals on Petri net formalism reveals that they are not suitable to formalize the informal aspects of the scenarios, to satisfy such properties as partiality and insensitivity to changes, and to handle the inherent abstraction and specialization of scenarios. HOONet, our proposed extensions of CPN towards object-orientation, is designed to bridge such gaps.

3.2 Basic definition of HOONet

A HOONet model is a representation of a "class" in object-oriented concepts. It consists of an identifier corresponding to a class name, Petri-nets representing the internal behaviors of the class, and a data dictionary declaring attributes and user defined functions.[12].

Definition 1 HOONet is a 3-tuple, HOONet = (O_p, ION, DD) satisfying the requirements below:

1. Object Identification Place O_P is a special place which is defined as a 4-tuple $O_P = (oip, iid, M_0, status)$, where
   - oip is the unique name of a HOONet model,
   - iid(instance identifier) is a unique identifier distinguishing the multiple instances of a class. iid is a tuple, $< oid, return >$, where oid is the unique object identifier obtained from the system, and $return$ is the address of the object which creates an instance.
   - M_0 is a function giving an initial marking to O_P place, and
   - $status$ is a flag variable to represent the specific state of O_P with a "pre" value and a "post" value.
2. Internal Object Net **ION** is a variant of CPN representing the internal behaviors of a HOONet model. It will be defined in Definition 2.
3. Data Dictionary **DD** is a part which declares variables, token types, and functions.

Figure 1 shows the general structure of the HOONet model. O_P contains the name oip of a class, and is also the starting and ending points of behaviors in a class.

When a token is given to O_P, a transition among the outgoing transitions of the O_P place is selected and fired. At the ending of the internal behaviors, the token behaviors flows through the border of ION, and the token comes into O_P place. The formal definition of ION is as follows:

Definition 2 The internal structure of HOONet is a 9-tuple, **ION** = $(\mathcal{P}, \mathcal{T}, A, \mathcal{K}, N, G, E, F, M_0)$, where

1. $\mathcal{P}$ and $\mathcal{T}$ are finite sets of places and transitions, respectively,

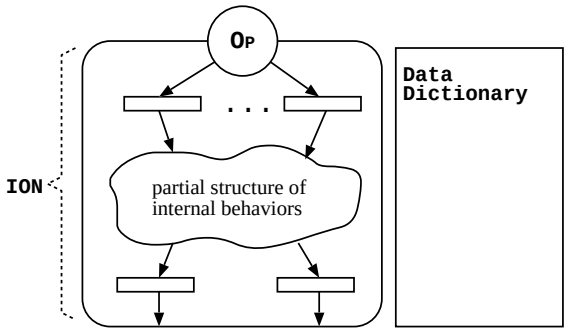

Figure 1: General structure of HOONet model

2. A is a finite set of arcs such that $\mathcal{P} \cap \mathcal{T} = \mathcal{P} \cap A = \mathcal{T} \cap A = \varnothing$,

3. $\mathcal{K}$ is a function mapping from $\mathcal{P}$ to a set of token types declared in DD,

4. N, G, and E mean the functions of a node, a guard, and an arc expression, respectively, which are the same as defined in CPN[14],

5. F is a special arc from any transition to O_P, and is notated as the border of a subnet ION, and

6. M_0 is a function giving an initial marking to a place.

DD defines the token types, variables, and functions for each HOONet models. Token types are classified into three categories: *primitive, abstract and user_defined*. The primitive type is the same as that in CPN, and the abstract type is a compound of primitive types. The abstract type is needed to express the representative states of detailed (sub) states or unrefined states of an abstraction. If a model is depicted with high-level information, the token type representing the states should be also represented with high-level information, since the token type expressing detailed information is not adequate to represent the states of abstract behaviors in the models. If a designer declares the token type in detail at the abstracted level, it is still acceptable. However, such representation for abstract states is not concise, and can cause changes in further refinements. The type of the abstract token is declared with the prefix "complex". This type is decomposed into several subtypes(primitive types) of tokens in refinement. The user_defined type means that it provides a way to extend type definition by creating new types. The definition of DD is very simple, and is written with textual grammar like the standard ML[30]. The syntactic grammar is given in Appendix A.

3.3 Behavior Description of HOONet

When we model the scenarios elicited from system requirements, we need ways to represent the interactions between (sub)systems and actors and to represent abstract information for system states

and its behaviors. The conceptual representations of the HOONet behaviors are shown in Figure 2. More detail definitions and their semantics are described in Appendix A and B.

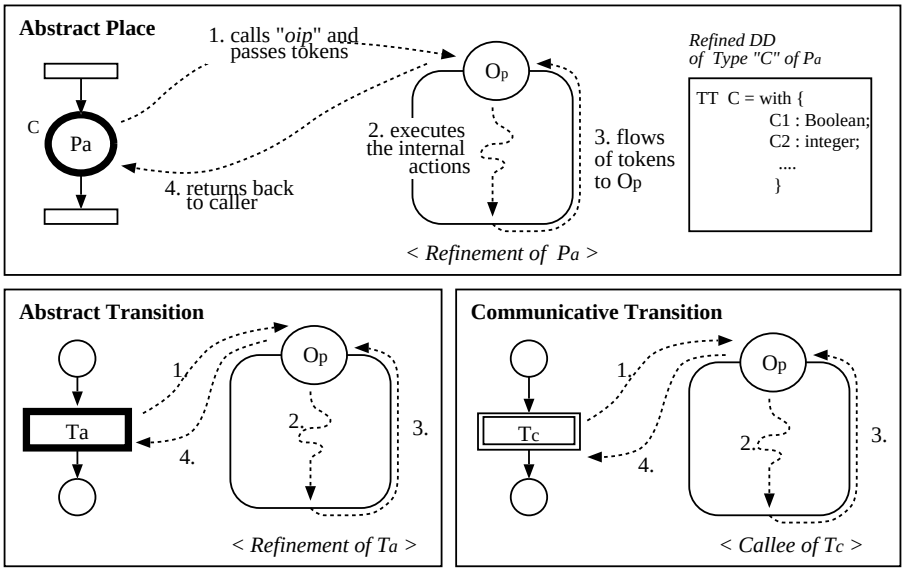

Figure 2: Conceptual representations of HOONet behaviors

The abstractions in HOONet are defined as abstract places, abstract transitions, and complex typed tokens. Abstract place(P_a) notated with bold-lined circle is a representation of abstract state, and abstract transition(T_a) notated with bold-lined rectangle is a representation of abstract action. These abstract components are refined into subnets depicting with detailed information in further modeling. Along with the refinement, complex typed tokens are also expressed with details using primitive typed tokens. In cases that abstractions have not been refined, their states are decided by the post conditions that are assigned to them.

Interactions between actors or between the system and actors are represented with communicative transitions(T_c) notated with double-lined rectangle in HOONet. This T_c invokes an object by calling a class, and passes its tokens to the invoked object. When the behaviors of the object have executed, the tokens are returned back to the T_c(caller transition). If a class which is not modeled yet is called, the actions of the caller transition are fixed in the post conditions similarly with that of abstraction.

Another behavior semantic that we define is on the inheritance of HOONet models. This is the same concept as the inheritance mechanism in the object-oriented paradigm. The behaviors of an inherited HOONet model are extended to include all behaviors of the inheriting super class(HOONet). The detail semantic of the inheritance will be shown in Appendix B, and its example is explained in the "specialization" part of Chapter 4.

4 Scenario Modeling using HOONet

4.1 Modeling Process

Figure 3 shows the modeling process which is based on the concepts of decomposition and refinement. This process can be performed iteratively and incrementally.

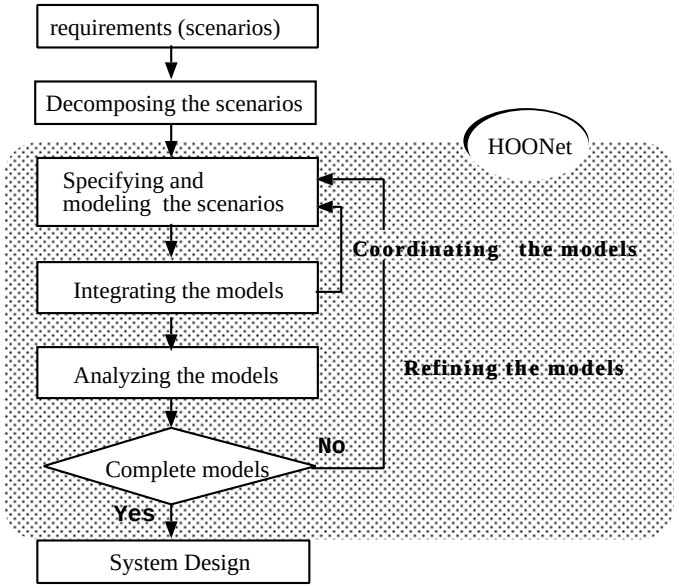

Figure 3: The modeling process of a system using HOONet

First, to model a system using HOONet, we decompose the scenarios of a system into actor-oriented behaviors and represent the decomposed scenarios in *script* forms. The scenarios represented by scripts are modeled using HOONet. Then, the behaviors of an actor which are modeled with multiple models of HOONet are combined into one HOONet model to represent the consistent behaviors of the actor. After the completion of the given scenario modeling, analysis of the models is performed with reachability graphs. To explain the modeling process of a system using HOONet, we give some scenarios of an example, a Car Rental Service(CRS) system, as follows:

S#1: Rental agency rents out a car according to a customer's request and receives a car returned by the customer.

S#2: To rent out a car, a rental agency checks the customer's credit, the required specification and rental period to decide the customer's good standing. If any of them is invalid, the rent-out will not be permitted.

S#3: A customer must take out an insurance policy to rent a car.

S♯4: The car should be returned when the rental period is expired or an accident occurs.

4.2 Decomposition of Scenarios

Behaviors of the given scenarios are decomposed into the behaviors of actors, and the behaviors are represented with a set of scripts. The script is defined as follows:

```
<scenario_no :  scenario_name>;
   {behavior_name,
      {coordinating_actor(c_actors):  participating_actor(p_actors),
      pre_condition,
      stimulus,
      response,
      post_condition
      }
   }
```

The next two scripts result from the decomposition of the above scenarios S♯1 and S♯2.

<S♯1: RENTAL_SERVICE>

behavior name	*c_actors : p_actors*	*pre_condition*	*stimulus*	*response*	*post_condition*
Rental	REAG: CUST	requested	renting	rent_ok	rented
Receive	REAG: CUST	rented	receiving	receive_ok	returned

<S♯2: TO_RENT>

behavior name	*c_actors : p_actors*	*pre_condition*	*stimulus*	*response*	*post_condition*
Renting	REAG: CUST	requested	check credit	bad | good	c_completed
		c_completed	bad	reject_rent	fail_rent
		c_completed	good	credit_ok	ok_credit
		ok_credit	check spec.	none | available	s_completed
		s_completed	none	reject_rent	fail_rent
		s_completed	available	available_ok	ok_available
		ok_available	check period	impossible | possible	p_completed
		p_completed	impossible	reject_rent	fail_rent
		p_completed	possible	rent_ok	rented

In the scripts above, the terms, "REAG" and "CUST" mean REntal AGency and CUSTomer, respectively. Intuitively, scenario S♯2 can be considered as the refinement of scenario S♯1 because the pre- and post-conditions of "Renting" behavior in script RENTAL_SERVICE are the same as the first pre-condition and the last post-condition of "Renting" behavior in script TO_RENT.

4.3 Modeling Scenarios Using HOONet

The scenarios represented with scripts are modeled with HOONet notations. Each item of the script definition is mapped into the features of HOONet as follows:

```
Mapping 1:  c_actor          ↦ name of an object
Mapping 2:  behavior_name    ↦ name of a method within an object
```

Mapping 3: pre_condition ↦ token value of input places

Mapping 4: stimulus ↦ a transition

Mapping 5: response ↦ expression of out arcs

Mapping 6: post_condition ↦ token value of output places

By the mapping rules above, the HOONet representation of scenario S♯1 can be obtained as shown in Figure 4. Two methods in the object REAG are represented with abstract transitions "renting" and "receiving."

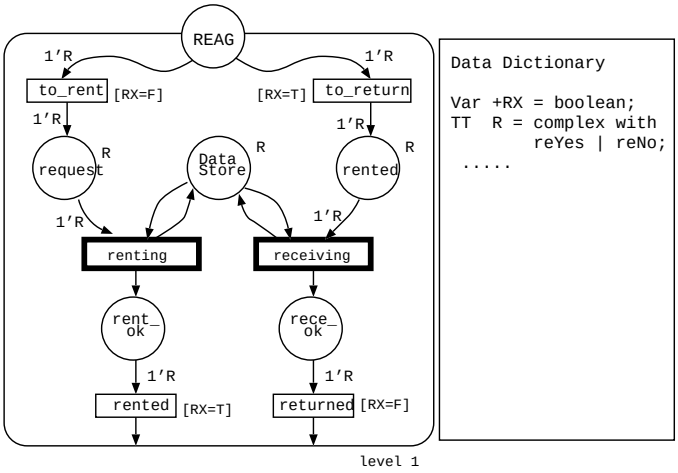

Figure 4: A HOONet model of the actor REAG

Representing abstraction When a specific behavior of an actor is included in multiple scenarios, the abstraction levels of the behavior can be described differently in each scenario. For example, scenarios S♯1 and S♯2 have different abstractions for the rental behavior of REAG actor. Figure 5 shows the HOONet model of the rental behavior in scenario S♯2 which resulted from a refinement of the abstract transition "renting" of the REAG(in Figure 4). The token type "R" in object REAG is also refined with some primitive types such as string and boolean in object RENTING.

To show the equivalence between an abstraction and its refinement, the constraints of the abstraction should be preserved in the constraints of the refinement. Such equivalence can be defined with the morphism between the nets. In HOONet models, the *sine qua non* conditions of the morphism are sufficient to show that the observable behaviors of the abstraction are equivalent to its refined net(shortly, RN). The constraints to define the observable morphism consist of three preservations,*token-type preservation, arc-expression preservation, and guard-condition preservation*, which are given as fol-

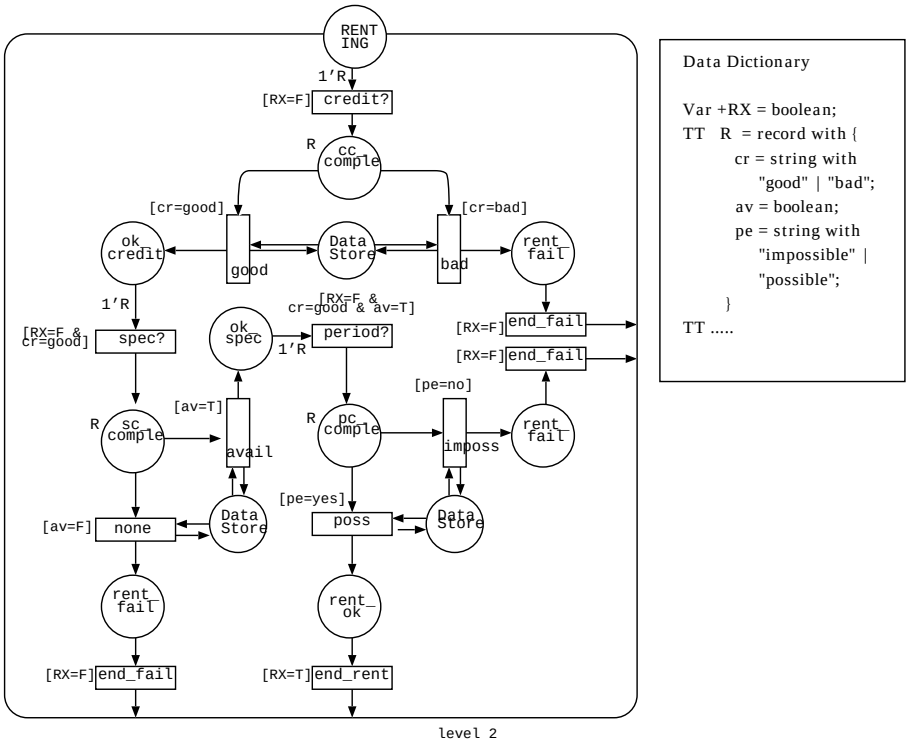

Figure 5: A refinement of the abstract transition "renting"

lows:

Definition 3 The **observable morphism** between abstractions and their refinements should satisfy the following requirements:

1. For token-type preservation,

   $$\forall p \in P_a,\ K(p) \subseteq TT_{RN}, \text{ and } \forall t \in T_a,\ \{K(\bullet t) \cup K(t\bullet)\} \subseteq TT_{RN},$$

   Note: The multi-set($K(p)$) of token types of every abstract place should be a subset of the set of all token types(TT_{RN}) defined in the Data Dictionary of RN. The multi-set of token types for incoming places and outgoing places of an abstract transition should also be a subset of TT_{RN}.

2. For arc-expression preservation,

   $$\bigcup_{\forall x \in \{P_a, T_a\}} Var(expr) \subseteq Var(O_{p_{RN}}) \subseteq V_{RN},$$

   Note: The set(V_{RN}) of all variables appearing in the incoming and outgoing arc expressions of an abstract transition should be a subset of the set of all variables defined in the Data Dictionary of RN.

3. For guard-expression preservation,

   $$\forall t \in T_a,\ G(t) \equiv \bigcup_{\forall t_i \in (O_{p_{RN}})\bullet} G(t_i).$$

Note: The evaluation of guard expression of an abstract transition should be equivalent with the evaluation of all guard expressions of outgoing transitions of $O_{P_{RN}}$.

Representing dynamic binding Operationally, dynamic binding may be regarded as a dispatching mechanism that acts like a case statement to select (dynamically) the appropriate behaviors in response to a value of a token. In HOONet, dynamic binding can occur in cases where abstractions are refined, or an object is called by a communicative transition. When the complex-typed token of an abstract component is passed to the refined net, the token should be mapped with more detailed tokens of the refined net depending on the token value. For example, let "acc_type" be a type to represent the extent of a car accident. In the high-level models, the declaration of the type "acc_type" will be defined as follows:

```
TT acc_type = complex with waste | critical | smolder | trauma;
```

If a car consists of six components such as engine, shaft, gear, radiator, wheel, and hood, the token types to represent the fault of each component, in the low-level models, will be declared as follows:

```
TT Engine = with disable | damage | unimpact;
TT Shaft = with broken | curved | abrasion | upright;
TT Gear = with collapse | defect | operate;
TT Radiator = with disrupt | fumes | normal;
TT Wheel = with crook | flat_tire | optimal;
TT Hood = with bend | scratch | slick;
```

When the abstract token "acc_type" in high-level models is refined, the token type can be bound with different subtypes according to its value. If the value of "acc_type" in high-level models is `waste`, the token type "acc_type" is decomposed into

```
TT acc_type = record with {engine = TT Engine with disable;
                           shaft = TT Shaft with broken | curved;
                           gear = TT Gear with collapse;
                           radiator = TT Radiator;
                           wheel = TT Wheel;
                           hood = TT Hood;
                           };
```

and, if the value of "acc_type" is `trauma`, it is decomposed into

```
TT acc_type = record with {engine = TT Engine with unimpact;
                           shaft = TT Shaft with upright;
                           gear = TT Gear with operate;
                           radiator = TT Radiator with normal;
                           wheel = TT Wheel with flat_tire | optimal;
                           hood = TT Hood;
                           };
```

Representing interaction The interaction between objects is achieved by communicative transition. When an object calls another object, an instance of the callee is generated, and the value of the variable *iid* of the instance is assigned with the concatenation of *oid* and *return*. Figure 6 shows the HOONet model for the rent behavior of a customer(CUST) and a copy of the REAG model in Figure 4.

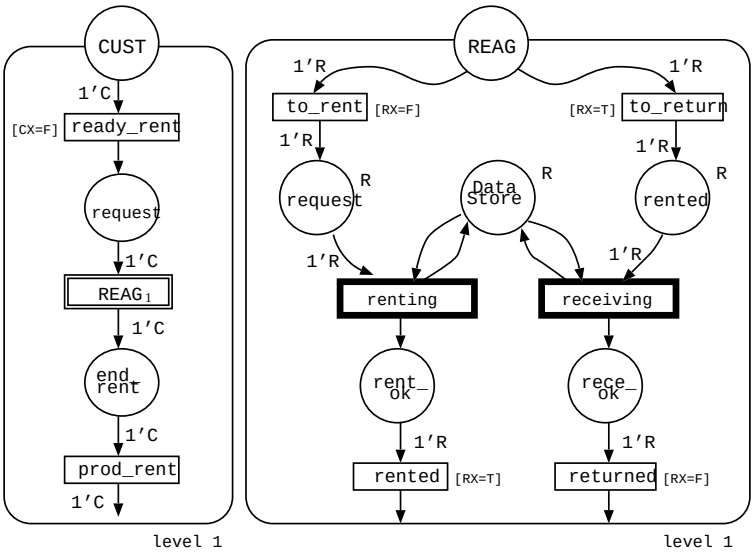

Figure 6: Caller object CUST and callee object REAG

When the object CUST requests a rent car through the communicative transition "$\texttt{REAG}_1$," the token is passed to the O_p of the object REAG. The transition "to_rent" is fired depending on the value of the input token. After the execution of rental behavior, the token is returned to the calling point of the caller object.

Representing specialization Some actors elicited from scenarios can be organized with a hierarchical structure. For instance, the object "VEH" depicting all rentable vehicles can be specialized with some objects such as cars, jeeps, trailers, and so on. This means that an object elicited from a scenario

can be described with the specialized or refined object in another scenario. This also means that the low-level object inherits the properties from the high-level object in the hierarchical structure. Figure 7 shows the HOONet models of "CAR" which inherits the properties from the object "VEH." The symbol to express inheritance is ":<". The inherited object "CAR" includes an additional behavior such as taking out an insurance policy for each rental car.

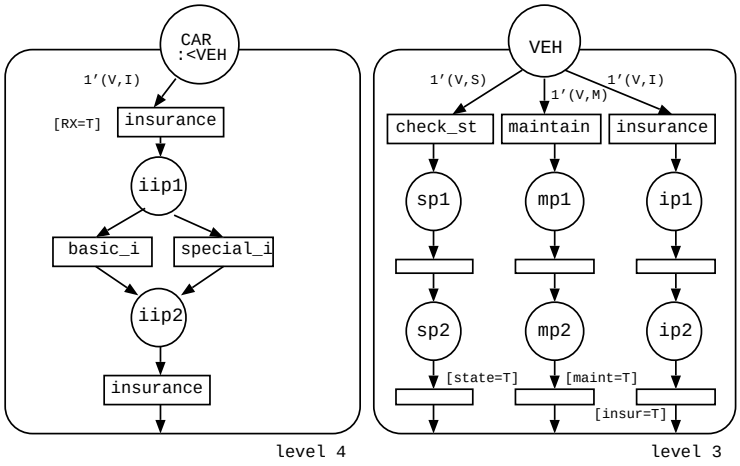

Figure 7: HOONet representation of inheritance

When an inherited object is instantiated, the Data Dictionary of the object is extended to include that of a high-level object, and the set of transitions which can be enabled with $E(O_P, t) < b >$ is also extended. Specialization by the inheritance mechanism should also support the overriding of inherited properties. The reasons to override the properties of high-level object are to specify a behavior that depends on the object, and to tighten the specification of a behavior. To support the overriding mechanism, it should be determined which property is supposed to be bound between the overridden property and the overriding property because overriding redefines a property with the same name, and it is not permitted to override the signature[1] or form of a feature. This decision is achieved by the scoping rule of the dynamic binding. The rule searches up the inheritance hierarchy to bind the property that implements a specific operation in the low-level object. In Figure 7, the added behavior `insurance` in the object "CAR" overrides the behavior `insurance` of the object "VEH."

5 Integrating Scenarios

The representation of global behaviors of a system can be shown by integrating HOONet models. This integration is performed by the following three steps.

[1] signature in object-oriented concepts means the number of arguments, the type of arguments, and the type of result value.

- Step 1: Unfolding abstract components of each HOONet model with refined models.
- Step 2: Integrating HOONet models by actor-oriented behaviors.
- Step 3: Coordinating behaviors of each actor.

5.1 Unfolding HOONet Models(Step 1)

The unfolding of HOONet models integrates the scenarios representing with the different abstraction levels. For example, scenario S♯2 in section 4 is a detailed representation of scenario S♯1. Thus, these two scenarios are integrated by the unfolding of S♯1 with S♯2. Figure 8 shows the unfolding of abstract transition "renting" in Figure 4 with its refined model in Figure 5.

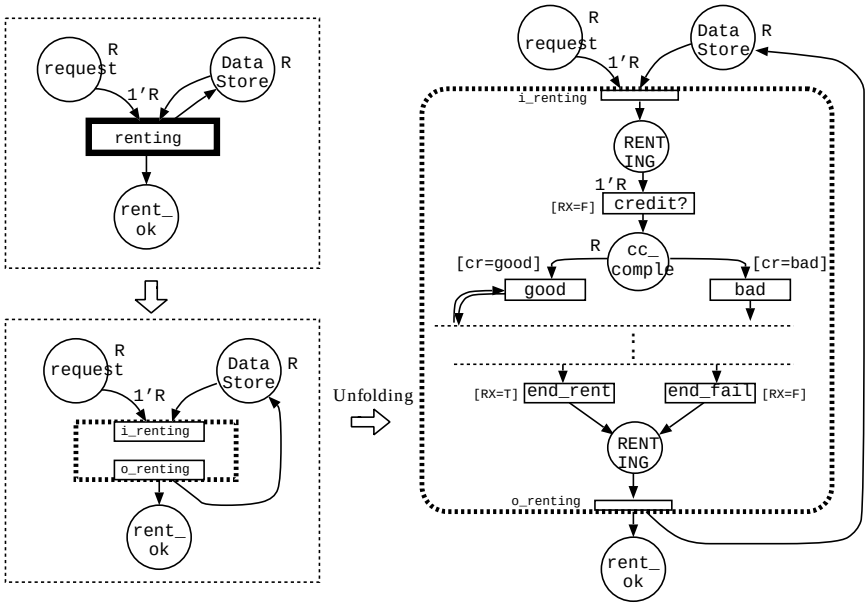

Figure 8: Unfolding of the abstract transition "renting"

As shown in Figure 8, abstract transition is split into two sub-transitions. The transition *i_renting* acts as the invocation of the refined behavior, and the transition *o_renting* is the ending of the invocation. Thus, the unfolding of abstract transition is achieved by insertion of the refined model between two sub-transitions. The refined model should also be unfolded before the insertion. The unfolding of the refined models is performed by duplicating the O_p as the exit after the removal of the body frame of HOONet.

Abstract behaviors without refinement are considered to be the missing parts of scenarios. The missing parts can be acquired from the users, and then can be modeled with HOONet. The features

such as abstract place, and communicative transition in HOONet model can be also unfolded. The unfolding mechanisms for these features can be referred in [12].

5.2 Integrating behaviors of an actor(Step 2)

The behaviors of an actor modeled by the multiple HOONet models can be integrated to represent the actor's behaviors in HOONet model. For example, a scenario includes a customer's behavior: renting a car, and another scenario includes the customer's behavior: returning the rent car. These two behaviors can be represented with a HOONet model containing the behaviors of the customer. The following steps are used to integrate multiple HOONet models into one model.

- Step 2.1: integrate data dictionaries of two HOONet models.
  - Step 2.1.1: removes the replicated variables, token types and functions.
  - Step 2.1.2: re-declares the global variables which are only used in the integrating models with local variables.
- Step 2.2: determine the firable transitions from O_p place of the target model.
- Step 2.3: merge the behaviors of an actor into the target HOONet model.

Figure 9shows the target HOONet model for a customer's behaviors generated by the steps above. HOONet model "$\texttt{CUST}_1$" depicts the rental behavior of a customer, and "$\texttt{CUST}_2$" depicts the returning behavior of the customer. These two models were integrated into an actor model "`CUST`". It is possible, while integrating scenarios, to remove the duplicated representations for the behaviors of an actor – for example, if the rental behavior of a customer is appeared in several HOONet models.

The target HOONet model which results from the integration of an actor's behaviors can gather a great deal of volume. Such a target model can be heuristically divided into two or more sub-models which are independent of each other – the intersection of places and transitions between the sub-models should be empty.

5.3 Coordinating behaviors between scenarios(Step 3)

An issue to be considered in the integration of actor's behaviors is the replicated representation. The replication appearing in some scenarios is categorized into four styles, as shown in Figure 10.

The replication style (a) can be removed in Step 2. The replication styles (b),(c), and (d) will be removed when two behaviors are observationally equivalent[18]. To verify the observational equivalence, we can consider the HOONet models as state-oriented transition models. Token transference from a set of places to another set of places, in general, means the state change of an actor. Let P

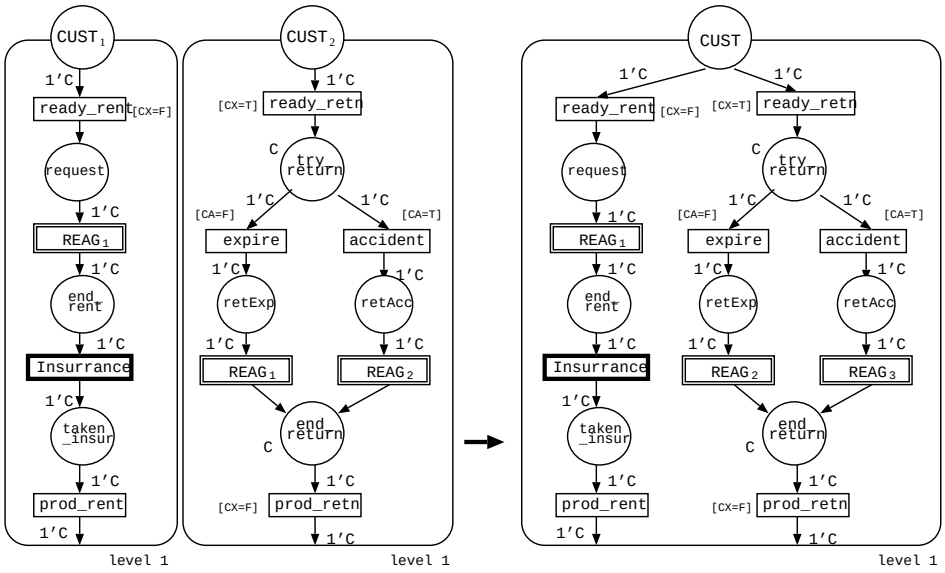

Figure 9: Behavior integration of an actor

and Q be the set of places representing the same states. If the P and Q are changed into P' and Q' by an action sequence α, respectively, and the P' and Q' are in the same state, we conclude these two behaviors, $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\alpha} Q'$, are identical.

The state-oriented transition model is the same as the reachability graph model of the HOONet model because the reachability graph enumerates all the possible states of an actor. The behavioral equivalence of two models can be verified by the bisimulation of two graphs after generating reachability graphs of the two models[18, 14]. Our algorithm to generate a reachability graph is described in Section 6. When each reachability graph of two HOONet models is generated, the observational equivalence for the two graphs is summarized as follows:

Definition 4 Let $\mathcal{P}$ and $\mathcal{Q}$ be the behaviors in two reachability graphs R_1 and R_2, respectively. And let P and P' be states in $\mathcal{P}$, Q and Q' be states in $\mathcal{Q}$. Two behaviors $\mathcal{P}$ and $\mathcal{Q}$ are **observation equivalence**($\mathcal{P} \approx \mathcal{Q}$) which should be satisfied as follows($\forall \alpha \in Act$ where Act is a set of actions α and a sequence t of actions $t = \alpha_1, ..., \alpha_n$):

1. *Whenever* $P \xrightarrow{t} P'$ *then, for some Q', Q* $\xRightarrow{\hat{t}} Q'$ *and* $P' \approx Q'$
2. *Whenever* $Q \xrightarrow{t} Q'$ *then, for some P', P* $\xRightarrow{\hat{t}} P'$ *and* $P' \approx Q'$

Note: P and Q are source states, and P' and Q' are destination states. Then, $\xrightarrow{t}$ exactly specifies the action sequence including τ actions, and $\xRightarrow{\hat{t}}$ specifies the action sequence excluding τ actions.

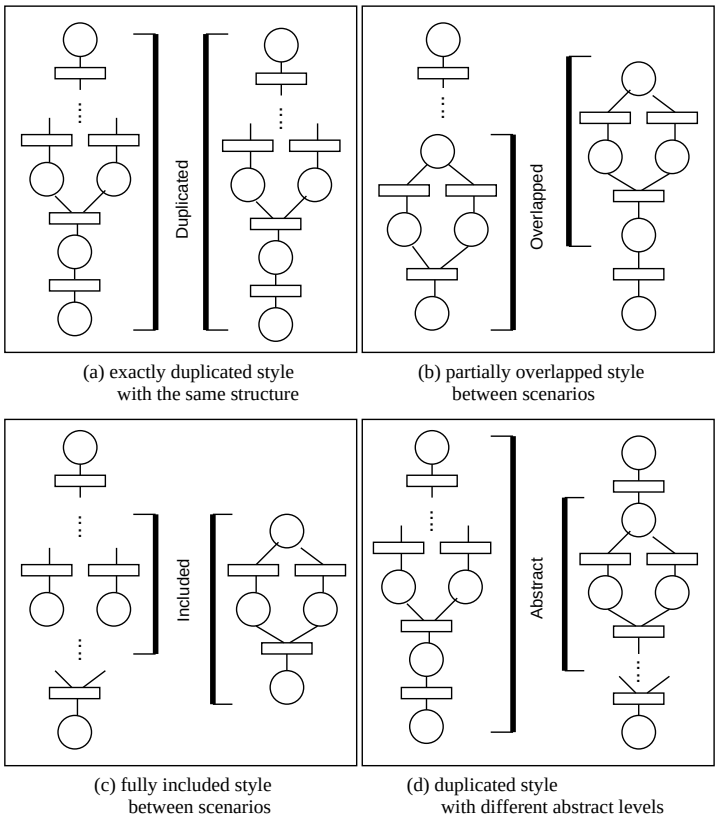

Figure 10: Replication styles of behaviors between scenarios

When P goes to P' by the action sequence with τ and then Q goes to Q' by the action sequence without τ, these two transition behaviors are observationally equivalent. □

For example, let's consider two scenarios of a customer's behavior for renting a car. These scenarios are described with action sequences of a behavior with different abstraction levels.

- scenario I: `ready_to_rent` → `rent_processing` → `processed_rent`
- scenario II: `ready_to_rent` → `credit_check` → `spec_check` → `period_check` → `processed_rent`

The above two scenarios I and II show that the action "rent_processing" corresponds to three actions "credit_check," "spec_check," and "period_check" in scenario II. These three check actions can be replaced with an action including τ because τ is a complementary action[18]. Thus, we can say that these two scenarios are representing the same behaviors which are observationally equivalent. However, the next three scenarios can be considered as different behaviors even if some of their action sequences are replaced by τ action.

- scenario III: `ready_to_rent` → `rent_processing` → `processed_rent`

- scenario IV: `ready_to_rent` → `credit_check` → `rent_fail`
- scenario V: `ready_to_rent` → `credit_check` → `spec_check` → `rent_fail`

From the three scenarios above, scenarios III and IV, and scenarios III and V are observationally equivalent, respectively by replacement of τ action. But, scenarios IV and V have different behaviors. Thus, the observational equivalence of two behaviors requires such a criterion as follows:

Constraints 1 *Observational equivalence of two behaviors appearing in different reachability graphs can be said when two action sequences occurring state transition are equivalent.*

Recipe: An action sequence in a reachability graph is an ordered set of transition labels. Transition labels are described with the combination of arc expressions, guard functions, and variables, which are declared in the data dictionary of HOONet models. In order to check the equivalence of two action sequences, we introduce the relabeling function[18]. Relabeling function extends and reduces the expression of an action by variable substitution. Thus,

relabeling function, f(x) = x'

means that all x are substituted with x'. This relabeling function is denoted with the symbol $a[x/x']$, where a is an action sequence. □

For example, because the token value `rent_FAIL` is in a token type `RX`, the type at the higher abstraction level is declared such as `TT RX = with rent_OK | rent_FAIL`. However, this declaration will be refined at the lower abstraction level as follows:

```
TT RX = record with { TT CREDIT = with good | bad;
                      TT SPEC = with available | none;
                      TT PERIOD = with possible | impossible;
                    }
```

Using the refined declaration, the token type `RX` of the higher abstraction level can be relabeled with $f(\texttt{RX}) = \texttt{CREDIT} \times \texttt{SPEC} \times \texttt{PERIOD}$. When two action sequences are equivalent after relabeling the variables of expression of action and replacing some actions with τ action, we can say that the two behaviors are observationally equivalent.

6 Analysis of HOONet Models

We have explained the mechanisms of modeling and integrating a system using the HOONet method. In this section, we discuss the reachability analysis method of HOONet models to check deadlock, liveness and nondeterminism of system behaviors.

6.1 Reachability graph generation

Reachability analysis, in general, is used to analyze several properties of Petri net models. This analysis method is based on a reachability graph which involves essentially the enumeration of all reachable markings. A reachability graph is defined as

$$RG = (V, A, N)$$

where, V is a set of nodes(vertices), A is a set of arcs(edges) such that $V \cap A = \varnothing$, and N is a node function defined from A into $V \times V$. That is to say, V means a set of states of a system and the state changes may occur along with the arcs.

Reachability graphs can be generated for each HOONet model. These graphs are used to verify the behavioral equivalence for some behaviors, and are also designed to represent the global system behavior. Because the semantics of HOONet have more or less different features than other high-level Petri nets such as CPN, the algorithm to generate a reachability graph of HOONet models is different from the conventional algorithms. The algorithm is shown in Algorithm 1. To increase the understandability of our algorithm, the definition of the next two functions precedes the algorithm.

- `GenNode`(M_1, M_2) is a procedure that creates a node(M_2), and connects the node(M_2) to a node(M_1) in the reachability graph; tags M_1 with a label "R", meaning a reachable node, and M_2 is tagged with a label "M", meaning its successors are not found yet.

- `AllGen`(M_1, M_i) is a procedure that finds all successors(M_i) of the node M_1, and connects the nodes(M_i) to the node(M_1); and then tags M_1 with "R", and all M_i is tagged with "M". Where, $i \geq 2$.

The conceptual mechanisms of the above two functions are shown in Figure 11.

For the meanings of all variables appearing in Algorithm 1, please refer Appendix A.

Algorithm 1 GenReachGraph(input:A_Set_of_HOONet_models, output:RG);

```
1.  Let the node which has initial markings, be the root
       of an RG; and tag this node with M
2.  While ({all nodes tagged with M} ≠ ∅)
    begin Switch(a node taged with M)
```

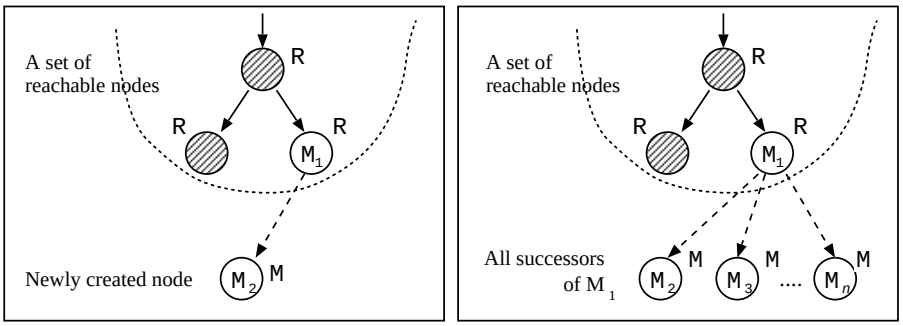

Figure 11: The conceptual mechanisms of the functions, GenNode and AllGen

```
    case O_p:  if (status = "pre") then AllGen(M_1, M_i);
               else if (iid.return = null) then Exit();
                    else GenNode(M_1, M_2) for node to return;
    case P_i:  AllGen(M_1, M_i);
    case P_a or T_a:  if (refine_state = "false")
                    then AllGen(M_1, M_i);
                    else GenNode(M_1, M_2) for O_p of refined net;
    case T_c:  if (target = "no") then AllGen(M_1, M_i);
               else if (c_type = "SYNC")
                    then GenNode(M_1, M_2) for O_p of called net;
                    else {AllGen(M_1, M_i) || GenNode(M_1, M_2) for O_p of called net};
end;
3.  End of Algorithm.
```

As the result of Algorithm 1, the set of all nodes tagged with "R" is a directed graph and also a reachability graph.

6.2 Incremental modeling and analysis

The reachability analysis of a system model using Petri nets, has a state explosion problem. To reduce this problem, the compositional analysis methods have been suggested. These methods decompose a system into subsystems or functional units, analyze reachability for each unit, and then compose their reachability graphs to a global graph[6, 31].

We suggest an incremental method to model and analyze a system using HOONet. Compared with other compositional reachability analysis methods, our method has some advantages: it can model and analyze a system whose requirements are not fully described or are not fully analyzed yet. The HOONet model may hide the internal behaviors of an object, and it has only one-entry and one-exit to interact with other objects, since the HOONet method provides the abstraction mechanism, the encapsulated objects, and the restricted public interface of each object. This *one-entry one-exit*

constraint permits to analyze the models by the compositional approach[29].

For example, the reachability graphs of the object "REAG" in Figure 4 and the object "renting" in Figure 5 can be separately generated, and can be merged into a global graph as logically shown in Figure 12.

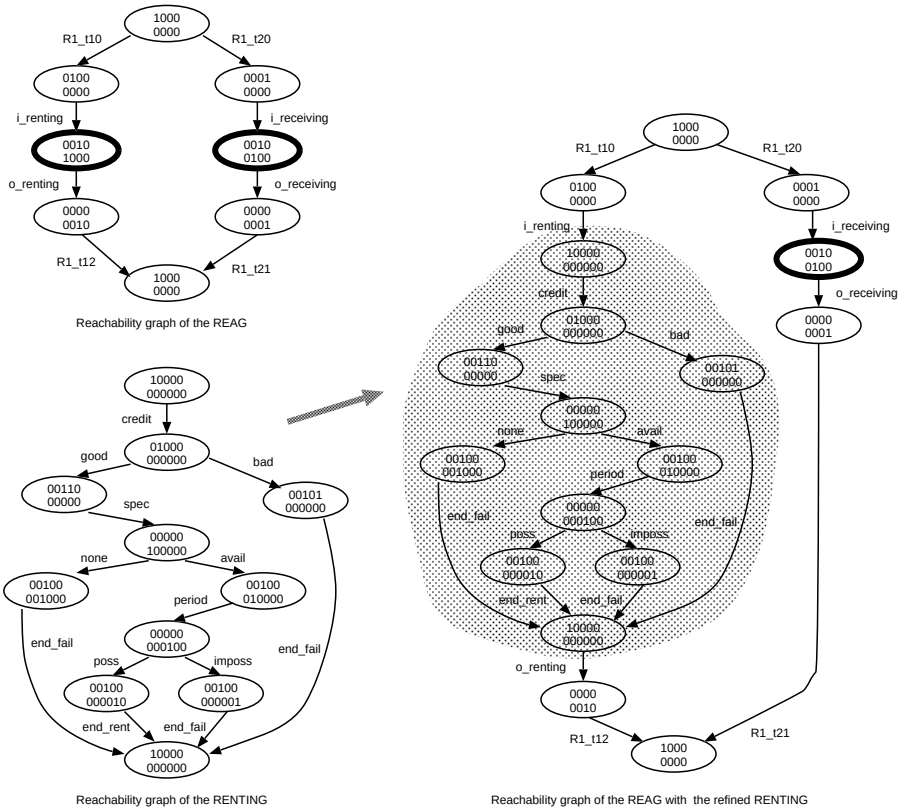

Figure 12: Generation of global reachability graph by merge

The steps to merge two reachability graphs are written in Algorithm 2. Let X be a set of all P_a, T_a, and T_c, and M_i be the preceding state of an element x_i in a given reachability graph.

Algorithm 2 MergeGraphs(input:RGs, output:Global_RG);

1. `Select a component` $x_i \in X$
2. `Using the algorithm 1, generate the reachability graph corresponding to the refined model of` x_i
3. `Inserts the reachability graph at the next location of` M_i

As the explanation of our HOONet modeling and anaysis above, scenario-based system requirements are incrementally modeled along with the abstraction hierarchy. Although the scenario modeling is not completed to analyze the given system requirements, we can perform the analysis of the HOONet

models. Also, if some portion of HOONet models are changed, we can only analyze the changed models.

Our incremental method is very useful in the design and analysis of complex and large scale systems because it is possible to generate the reachability graph only for the parts of interest.

6.3 Consistency and Completeness

Various Petri nets analysis techniques can be applied to HOONet models to detect errors such as inconsistency or incompleteness of the given requirements. In this section, we point out the cases occurring inconsistency and incompleteness from the given requirements, and we give further explanation about how these cases appear on the reachability graph of HOONet model, and how these cases can be removed from the requirements.

Inconsistency can be occurred in a case that an event in scenario prohibits another event from occurring. The transition which can be triggered from the prohibited event cannot be enabled. Also, in a case that only a specific event sequence occurs regardless of the inputs values, for example the guard conditions' mismatch, we can say that this case is inconsistent. These cases mean that some transitions represented in HOONet model are never enabled. Thereby those transitions are not included in the nodes of the reachability graph, and the set of never-enabled transition can be gained to generate the reachability graph.

Another inconsistency case of scenario is caused by deadlock. Deadlock can occur when concurrent behaviors in scenarios mutually wait to occur a specific event. In this case, any transitions in HOONet models do not exist on the enabled state. Thereby generating the reachability graphs for HOONet models will be blocked.

When some scenarios have partially missed the descriptions of the requirements, the behaviors of those scenarios will be blocked without further proceeding. This incompleteness occurs, in general, when the negative case of the behaviors was not described. When events triggering the negative(missed) behavior occur, any transition that can be enabled in HOONet model does not exist. Thus, the reachability graph for these portions will not be generated, and the graph will not reach the final state.

There are some similar cases causing inconsistency and incompleteness of the given scenarios. These cases also can be revealed in similar forms with the cases above. Therefore, we can check those flaws by the general analysis techniques using a reachability graph of Petri nets.

7 Conclusion and Future Work

In software and system engineering areas, management of scenarios is emerging as a useful technology for specifying, modeling, testing a system, and managing the change of system information. Practically, research and applications of scenario management have been done in CREWS, ESPRIT projects[1].

Our paper focuses on providing the solutions of two problems which can appear in the process of scenario-based modeling of complex and large-scale systems; One is how the different abstraction levels of a behavior included in multiple scenarios are coordinated, and the other is the possibility of modeling the system requirements that are not fully analyzed yet. To solve these problems, we suggested HOONet which is an object-oriented high-level Petri net and the modeling procedures of a system using HOONet. In the process of scenario integration, we also suggest the ways of identifying the replicated and missed parts of system behaviors, coordinating the different abstraction levels of behaviors and analyzing the reachability of HOONet models using incremental approach.

Our results can provide the adequate representation of system behaviors as well as the completeness and consistency of scenarios integration. Especially, the HOONet method copes with the difficulties of the two issues mentioned above - abstraction level coordination and incremental modeling and analysis, which have not been handled in the existing proposals. We believe that these results can contribute for the modeling and analyzing a system using scenarios in large and complex industrial applications.

While our research offers those improvements mentioned above, in the scenario-based software modeling, there are some issues that are worthy of further research. First, software tools to support HOONet method are needed because the productivity gains are limited when applying our method manually. Thus, we are now developing the HOONet tools using $VisualCafe^{TM}$ and $Java$ language. Second, HOONet formalism itself could be extended. The area for the extension will include the support for timing analysis. Finally, applying the HOONet formalism to various software engineering domains can be studied. Reuse of the HOONet models and change management of requirements will become other research domains where our HOONet formalism can show its ability.

References

1. C. B. Achour, Linguistic Instruments for the Integration of Scenarios in Requirements Engineering, CREWS Report 97-09, appeared in *Proceedings of the Third Int'l Workshop Requirements Engineering: Foundation for Software Quality*, June, 1997.

2. E. Battiston, F. Cindio and G. Mauri, Modular Algebraic Nets to Specify Concurrent Systems, IEEE Transaction on Software Engineering, Vol.22, No.10, 1996, pp. 689-705

3. H. Ben-Abdallah and S. Leue, MESA: Support for Scenario-Based Design of Concurrent Systems, Technical Report 97-12, Dept. of Electrical and Computer Engineering, University of Waterloo, Canada, 1997.

4. D. Beringer, Modeling Global Behavior with Scenarios in Object-Oriented Analysis, Ph.D Thesis, Ecole Polytechnique Federal de Lausanne, Department d'Informatique, Switzerland, May, 1997.

5. O. Biberstein, D. Buchs and N. Guelfl, Modeling of Cooperative Editors Using COOPN/2, Proceeding of the Int'l Workshop on Object-Oriented Programming and Models Concurrency(OOMC'96), Osaka, Japan, June 1996, pp.87-105

6. S. Cheung and J. Kramer, Context Constraints for Compositional Reachability Analysis, ACM Transaction on Software Engineering and Methodology, Vol.5, No.4, Oct., 1996, pp.334-377.

7. A. Cockburn, Goals and Use Cases, Journal of Object-Oriented Programming, September, 1997, pp.35-40

8. Y. Deng, S.K. Chang and X. Lin, Executable Specification and Analysis for the Design of Concurrent Object-Oriented Systems, International Journal of SEKE, Vo.4, No.4, 1994, pp.427-450

9. J. Desharnais, et al, Integration of Sequential Scenarios, IEEE Transaction on Software Engineering, Vol.24, No.9, September, 1998. pp. 695-708

10. M. Glinz, An Integrated Formal Model of Scenario Based on Statecharts, Proceedings of ESEC'95, Spain, September, 1995, pp.254-271

11. Pei Hsia, Jayarajan Samuel, et al, Formal Approach to Scenario Analysis, IEEE Software, March, 1994, pp.33-41

12. J.E. Hong and D.H. Bae, HOONets: Hierarchical Object-Oriented Petri Nets for System Modeling and Analysis, Technical Report CS-TR-98-132, Dept. of Computer Science, KAIST, November 1998, http://cs.kaist.ac.kr/library/tr/index.html

13. I. Jacobson, and et al, Object-Oriented Software Engineering: A Use Case Driven Approach, 1992, Addison Wesley.

14. K. Jensen, Coloured Petri Nets: Basic concepts, Analysis methods, and Practical use, Vol 1. 2., Springer-Verlag, 1992

15. LOOPN++: A New Language for Object-Oriented Petri Nets, Technical Report R94-4, Networking research group, University of Tasmania, Australia, 1994

16. W.J. Lee and et al, Integration and Analysis of Use Case Using Modular Petri Nets in Requirement Engineering, IEEE Transaction on Software Engineering, Vol.24, No.12, Dec., 1998, pp.1115-1130.

17. F. Lustman, A Formal Approach to Scenario Integration, Annals of Software Engineering, Vol.(3), 1997, pp. 255-271

18. R. Milner, Communication and Concurrency 1989, Prentice Hall Ltd.

19. H.W. Nissen, R. Motschnig-Pitrig, and M. Jarke, View-Directed Requirements Engineering: A Framework and Metamodel, CREWS Report 97-11, appeared in Proceedings of the SEKE'97, June, 1997. pp.366-373

20. A. Perkusich, M. Perkusich and S.K. Chang, Object Oriented Design, Modular Analysis, and Fault-Tolerance of Real-Time Control Software Systems, International Journal of SEKE, Vol.6, No.3, 1996, pp.447-476

21. A. Perkusich and J.C.A. Figueiredo, G-Nets: a Petri Net based Approach for Logical and Timing Analysis of Complex Software Systems, Journal of Systems and Software, Vol(39), 1997, pp.39-59

22. W. Resig, Petri Nets: An Introduction, Berlin, Springer-Verlag, 1985

23. B. Regnell, Improving the Use Case Driven Approach to Requirement Engineering, Proceedings of ISRE'95, York, March 1995, pp.40-47

24. C. Rolland, C.Ben Achour, et al, A Proposal for a Scenario Classification Framework, CREWS Report 96-01, City University, London, 1996

25. J. Rumbaugh et al, Object-Oriented Design and Modeling, Prentice-Hall, Inc., 1991

26. V.P. Sliva, T. Murata and S.M. Shatz, Protocol Specification Design Using An Object-Based Petri Net Formalism, International Journal of SEKE, Vol.9, No.1, 1999, pp.97-125

27. S. Some, et al, From Scenarios to Timed Automata: Building Specifications from Users Requirements, Proceedings of APSEC'95, Australia, December 1995, pp.48-57

28. S. Some, R. Dssouli and J. Vaucher, Toward an Automation of Requirement Engineering using Scenarios, Journal of Computing and Information, Vo.2, No.1, 1996, pp.1110-1132

29. I. Suzuki and T. Murata, A Method for Stepwise Refinement and Abstraction of Petri Nets, Journal of Computer and System Sciences, 27, 1983, pp.51-76

30. J. Ullman, Elements of ML Programming, Prentice-Hall Inc., 1998

31. W.J. Yeh and M. Young, Compositional Reachability Analysis Using Process Algebra, Proceedings of Symposium on Testing, Analysis, and Verification, 1991, pp.49-59

Appendix: Syntax and Semantic Definition of HOONet

We define syntax and semantics of HOONet excluded in chapter 3. This appendix contains the formal definitions of HOONet structures for abstraction and interaction. Also, the behavioral semantics including enabling and firing of HOONet are formally defined.

A Definition of HOONet structures

Definition of Places: A set of places in HOONet is defined as $\mathcal{P} = \{P_i, P_a\}$, where

1. P_i is a basic place to represent local states of the system, the same as primitive place in basic Petri nets[14].
2. An abstract place, $P_a = (pn, refine\_state, action)$ represents the abstract states, where
   - pn is the name of an abstract place,
   - $refine\_state$ is a flag variable to denote the refinement of an P_a, and
   - $action$ is the static reaction imitating the internal behaviors of an abstract place.

Note: The variable $refine\_state$ is a boolean variable. If the value of $refine\_state$ is "true", the refined net of an P_a is available to be referred from other classes, and a "false" value of $refine\_state$ means that the refined net is not modeled yet. The $action$ is an algebraic expression corresponding to a post-condition which must be satisfied when the value of $refine\_state$ is "false".

Definition of Transitions: A set of transitions in HOONet, $\mathcal{T} = \{T_i, T_a, T_c\}$, where

1. P_i is a basic transition in basic Petri nets[14].
2. An abstract transition $T_a = (tn, refine\_state, action)$, where
   - tn is the name of an abtract transition,
   - $refine\_state$ has the same meaning as that in the definition of abstract place, and
   - $action$ is a static reaction imitating the internal behaviors of abstract transition.
3. A communicative transition $T_c = (tn, target, c\_type, action)$ is a transition to interact between classes, where
   - tn is the name of T_c,
   - $target$ is a flag variable to denote whether the callee of T_c was modeled("yes" value of the variable $target$) or not("no" value),
   - c_type is also a flag variable representing whether the interaction of T_c is synchronous("SYNC" value) or asynchronous("ASYN" value), and
   - $action$ is a static reaction to imitate the behavior execution of the callee.

DD defines the token types, variables, and functions for every HOONet. The definition of DD is very simple and is written with textual grammar like the standard ML[30]. The syntactic grammar is given as follows:

Syntactic Grammar of Data Dictionary:

- Declaration of token types

  {+}TT *token_type* = `with` *type_value* + { "|" | *type_value* } $^{0}_{n}$
  {+}TT *token_type* = *primitive_type* `with` [*constraints*]
  {+}TT *token_type* = *complex_type*.
  {+}TT *token_type* = `record with` {*token_type* = *primitive_type;*}$^{1}_{n}$ +
  {*token_type* = *complex_type;*}$^{0}_{n}$
  {+}TT *token_type* = `record with` {*token_type* = `record with`
  {*token_type* = *primitive_type;* | *token_type* = *complex_type;* |
  token_type = *token_type;*}}$^{0}_{n}$
  primitive_type = {*integer* | *real* | *boolean* | *string* }
  complex_type = `complex` *token_type;*

- Declaration of variables

  {+}`Var` *variable_name* : {*token_type* | *primitive_type* | *comp lex_type* };

- Declaration of functions

  {+}`Fun` *function_name* (< *parameter_list* >) =
  {< *pre_condition* > + < *post_condition* > | < *expression* > }
  - *parameter_list* = {*variable* | *place_name* | *transition_name*} + {": " + *type*}
  - *pre_condition* | *post_condition* = *boolean-expression*
  - *expression* = {*arithmetic_statement* | *case_statement* | *if_statement* }

- Redefine the token type, the variables and the functions

  `Redefine` {`TT` *token_type* | `Var` *variable_name* | `Fun` *function_name*} :
  {*declararion of* {`TT` | `Var` | `Fun`}};

- Static reactions of P_a, T_a, and T_c

  P_a(*place_name*) = {*(condition of flag_variables) : post_conditions*}
  T_a(*transition_name*) = {*(condition of flag_variables) : post_conditions*}
  T_c(*transition_name*) = {*(condition of flag_variables) : post_conditions*}

  - *condition of flag_variables* | *post_conditions* = *boolean-expression*

In the definition above, the notation "+" menas that the declaration can be globally referenced. The "with" phrase is to represent the constraints imposed to primitive type of token, for example, "integer with [1..100]".

B Behavior Semantic of HOONet

Before the formal description of the behavior semantic - that is the enabling and firing semantics of HOONet, we give some notations for an unambiguous and simple explanation of the description.

- $\forall t \in \mathcal{T}$, Var(t) is a set of variables of transition t. Var(*expr*) means a set of variables in the expression *expr*.
- $E(x_1, x_2)$ is an expression of (x_1, x_2), where $\forall (x_1, x_2) \in (\mathcal{P} \times \mathcal{T} \cup \mathcal{T} \times \mathcal{P}) : E(x_1, x_2) = \Sigma_{a \in A(x_1, x_2)} E(a)$.
- A binding b of a transition t is a substitution that replaces each variable of t with a token type. It is required that each token be of the correct type and the guard evaluates to true. Thus, $E(p,t) < b >$ denotes the evaluation of the expression from p to t in the binding b, and yields the multi-set of token types, which are removed from p when t occurs with the binding b.
- $\forall p \in \mathcal{P}$, the function $\mathcal{K}(p)$ is a mapping of each p to a set of token type TT (described in DD). M(p) is a marking function which reflects the token types and number of them with specific values to the place p.
- Let x be a place or a transition. $\bullet x$ is the input to x and $x\bullet$ is the output from x. This is formally defined as $\bullet x = \{y \in \mathcal{P} \cup \mathcal{T} | (y, x) \in A\}$ and $x\bullet = \{y \in \mathcal{P} \cup \mathcal{T} | (x, y) \in A\}$, where A is a set of arcs.

Behavior of O_p Place: The behaviors of Petri nets are depicted with the changes of token markings. The marking of O_p is in cases of initial marking and called marking. The called marking means that an O_p is called by other objects to perform a behavior. This can be separately considered with two cases: One is at the time when an O_p is just called[$O_p.status$ = "pre"], and the other is when the behavior of a method has been executed[$O_p.status$ = "post"]. The behavior, when an O_p is initially marked or is just called, is the binding of a transition with $E(O_p, t) < b >$. The behavior when $O_p.status$ = "post", is return back to calling object, or to terminate the behavior. These behaviors are defined as follows:

$$\begin{cases} \forall t \in O_p\bullet,\ E(O_p,t) < b > & \text{if } status = \text{"}pre\text{"} \\ M(X\bullet) \text{ for } X = O_p.iid.return & \text{if } status = \text{"}post\text{"} \end{cases}$$

If the value of variable *return* of an object is "null" , the behavior of the object is terminated because the object corresponds to the main() procedure.

Behavior of Abstract Components: Abstract components have different behaviors depending on whether those components are refined or not. If an abstract component is refined[$refine\_state =$ "$true$"], the tokens of the component are transfered to its refined object. Otherwise[$refine\_state =$ "$false$"], the static reaction of the component is performed(i.e., post-conditions are fixed), and then it proceeds. The behaviors of an abstract place(P_a) are defined as follows:

$$\begin{cases} M(O_p) \text{ for } P_a.pn = O_p.oip & \text{if } refine\_state = \text{"}true\text{"} \\ R(action);\ \forall t \in P_a\bullet, E(P_a, t) < b > & \text{if } refine\_state = \text{"}false\text{"} \end{cases}$$

Where $R(action)$ means the evaluation of the static reaction of abstract components. The behaviors of the abstract transition(T_a) are defined as follows:

$$\begin{cases} M(O_p) \text{ for } T_a.tn = O_p.oip & \text{if } refinr\_state = \text{"}true\text{"} \\ R(action);\ +M(T_a\bullet) \wedge \text{-}M(\bullet T_a) & \text{if } refine\_state = \text{"}false\text{"} \end{cases}$$

Where the term, $+M(T_a\bullet)\wedge$ -$M(\bullet T_a)$ means the addition of tokens to output places and deletion of tokens from input places by the firing of the T_a.

Behavior of Communicative Transition: Communicative transition(T_c) represents the interactions between objects(HOONet models). The behavior of a T_c depends on two factors: Is the called object modeled ? and What is the type of the communication mechanism ? If the callee was not modeled, the static reaction of the T_c is performed. Otherwise tokens are transfered to the O_p of the callee. The behaviors of synchronous communication are defined as follows:

$$\begin{cases} M(O_p) \text{ for } T_c.tn = O_p.oip;\ \text{-}M(\bullet T_c) & \text{if } target = \text{"}yes\text{"} \\ R(action);\ +M(T_c\bullet) \wedge \text{-}M(\bullet T_c) & \text{if } target = \text{"}no\text{"} \end{cases}$$

The behaviors of T_c with asynchronous communication type transfer tokens to the O_p of the called object and $T_c\bullet$ both. The execution results of the called method are stored in the global variable to be used in later operations. The following is the behavior semantics of asynchronous communication.

$$\begin{cases} M(O_p) \text{ for } T_c.tn = O_p.oip;\ +M(T_c\bullet) \wedge \text{-}M(\bullet T_c) & \text{if } target = \text{"}yes\text{"} \\ +M(T_c\bullet) \wedge \text{-}M(\bullet T_c) & \text{if } target = \text{"}no\text{"} \end{cases}$$

Behavior of Inherited HOONet model: Inheritance is the sharing of all properties among HOONet models based on a hierarchical relationship. When an inherited model is instantiated, the Data Dictionary of the model is extended to include that of higher-level model, and a set of transitions which can be enabled with $E(OIP, t) < b >$ is also extended. An issue to be considered in the inheritance mechanism is overriding the properties or behaviors of a higher-level model by modification.

Inheritance from higher-level models satisfies the requirements as belows. Let I, S, and N be an inherited HOONet, a higher-level inheriting HOONet, and the newly defined and overriding features, respectively.

1. $TT_I = TT_S \cup TT_N$,
2. a set of enabled transitions, $ET_I = \{t_i | \forall t_N,\ E(o_{p_I}, t) < b >\} \cup \{t_j | \forall t_S,\ E(O_{p_S}, t) < b >\}$, and
3. for $t_i \in \mathcal{T}_N\ \wedge\ t_j \in \mathcal{T}_S,\ \exists t_i, t_j \in ET_I\ \mapsto\ ((t_i, b) \in Y\ \wedge\ (t_j, b) \notin Y)$.

□