Generating Petri Net-Based Behavioral Models From Textual Use Cases and Application in Railway Networks

Zuohua Ding, Member, IEEE, Mingyue Jiang, and MengChu Zhou, Fellow, IEEE

Abstract—A software system’s requirements are often specified by textual use cases due to the latter’s concrete and narrative style of expressions. However, they have limitation in the synthesis of the system behavior since they have a poor basis for the formal interpretation. Existing synthesis techniques are either largely manual or focus on the use case interactions. We present a framework from a model-based point of view to automatically synthesize system behavior from textual use cases to a Petri net model. The generated net model can well describe component module interactions and thus can be used to check the requirement properties. The function of Send-Railway-Emergency-Call of European Integrated Railway Radio Enhanced Network is used to show the proposed method. Moreover, the experimental results on a set of examples demonstrate the effectiveness of the proposed method.

Index Terms—Textual use case, natural language processing, Petri net, model transformation, railway network.

I. INTRODUCTION

SOFTWARE requirements are often given in either graphical or textual formats. The graphical formats are typically based on Message Sequence Charts (MSCs) and Unified Modeling Language (UML) sequence diagrams [19], while the textual formats are based on natural languages [18]. Due to their concrete, narrative style of expressions, textual use cases in a natural language [4] are very effective for eliciting software requirements, and are easily understood by users and service suppliers. However, they lack formal syntax and semantics, and thus have a poor basis for formal interpretation of the behavior, which may increase the possibility of errors being introduced in the interpretation process. A formal requirement specification with mathematical precision and rigor can reduce interpretation errors. Formal analysis can detect errors such as inconsistency and incompleteness in requirements specifications. With the support of some tools, the detection process can be automated.

Recently, efforts have been made to transform textual use cases to formal presentations. For examples, they are transformed into first-order predicate logic specification [8], [9], Pro-case (Protocol use case) [24], a Two-Level Grammar knowledge base, which is subsequently translated into Vienna Development Method [21], temporal logic [31], process algebra, Circal [7], and executable live sequence charts [12]; Certain linguistic patterns are converted into first order logic [11], [35]; Agent-Oriented (AO) systems requirements are translated into a formal specification in Communicating Sequential Processes (CSP) [33].

Respecting these studies, we have identified some of their shared issues:

- Are there any formal models that can preserve the requirement properties?
- Is it possible to automatically build a formal model from informal descriptions?

To address the above issues, this paper presents a framework from a model-based point of view to automatically transform textual use cases to Petri net-based behavioral models. We raise some constraints on their writing to reduce the vagueness. Our restricted language style falls into the language categories in [2], [11], [20].

Our net model is a subclass of Petri nets. In it, each subsystem is described by a Petri-net-state-machine, called a closed process net, and the interactions among the subsystems are through a rendezvous mechanism. The reasons we choose such nets are that (1) they allow modeling of state-based concurrent systems and are well suited to the specification of synchronization issues; (2) they enjoy a variety of analysis and simulation tools to check the properties; and (3) they have close connection to such high level languages as Calculus of Communicating systems (CCS) [26], CSP [16] and Ada, which are also rendezvous-based; and (4) they can be used for the performance analysis [6] and static analysis [5].

The following is the sketch of the transformation process. We first construct Source Metamodel from textual use cases, and Behavior Target Metamodel for the target Petri net, and then define rules to transform a source model to a target one. The model transformation relies on the model transformation platform ADT (ATL Development Tools) provided by Eclipse."
Fig. 1 illustrates part of our model generation process. A collection of use cases is mapped to a use case source model Ma. Ma is then mapped to the behavior target model Mb by transformation model Mt. Here Ma and Mb are written in ecore while Mt is written in Atlas Transformation Language (ATL). The Source Metamodel MMA and Target Metamodel MMB are described by UML. Ma conforms to MMA, Mb conforms to MMB, and Mt conforms to the transformation metamodel MMT. The transformation model is built on top of the model transformation platform ADT. In particular, ADT is used to construct MMT.

Our method has the following advantages:

- The transformation process is automatic. The use case metamodel can be automatically configured as the input to the model transformation, which is the key step in the transformation. By using some Natural Language Processing (NLP) technique, we can extract the information from textual use cases and automatically configure the use case metamodel. Thus, the synchronization exists between textual use cases and a Petri net, which means a modification of textual use cases will reflect immediately in their corresponding Petri net.
- Requirements can be checked and viewed. Since the target Petri net model has both formal and informal definitions, it can be used for the requirement property checking and the user interface. Due to its formal form, we can use model checking tools, such as SPIN, to check the properties such as completeness and consistency. Because of its informal form, we can use it to display the detected errors graphically, and can help users understand them easily.

Section II defines a new use case metamodel; Section III presents a new behavior metamodel based on Petri nets; In Section IV, we configure the attributes of use case metamodel to obtain a source model based on the values extracted from textual cases; In Section V, we configure the behavior metamodel to derive a target metamodel based on the values from model transformation; In Section VI, we describe a prototype supporting the model transformation and illustrate our method with a real example; Section VII discusses the related work; and Section VIII concludes the paper.

II. USE CASE METAMODEL

In order to automatically process textual use cases from a model-based point of view, we need a use case metamodel.

Fig. 2. Subject-verb-[object], where S stands for a sentence, NP for a noun phrase, VP for a verb phrase, NNP for a singular proper noun, VBZ for a 3rd person singular present verb, and NN for a singular or mass noun.

Existing metamodels either contain too little information for our purposes such as UML use case models or do not provide design information such as activity diagrams [13].

A. Textual Use Case

Use cases can be written via charts, programming languages, or text. Since they serve as contracts among people who may not possess domain knowledge, simple text is a preferred choice. This work chooses the format [4] to write the use cases. Their primary elements are listed as follows:

Use Case | System under Discussion |
Primary actor | Scope |
Main scenario: 1. step 1, 2. step 2, . . . , n. step n |
Variation | Extension |

There are two kinds of System under Discussion (SuD): one that represents the system itself, denoted as S-SuD, and others that represent the subsystems. The following sentence styles are used to write the sentences in the steps of a use case:

i) In the main scenario, extension and variation, the sentence in every step describes only one activity of the actor (primary actor, SuD). The sentence can be specified in the following structures.

- subject+verb+[object]
- subject+verb1+object1+to+verb2+[object2]
- subject+verb+object1+to/from+object2
- subject+verb+object+adjective
- subject+verb+object+present participle
- subject+verb+object+past participle

where “+” is a connector used between two elements, which helps to build a sentence format. Each type of sentences corresponds to a syntax tree. For example, the first one corresponds to a syntax tree in Fig. 2.

ii) The names of users, subsystems, and the external systems start with capital letters, while all other words start with small letters.

iii) For two sentences that describe the same activity in two different use cases, they should have the same description.
Remark 2.1: Our restricted sentence styles belong to the language categories used in [2], [11], and [20]. Many examples in the literature can be described with our sentence styles, such as Car Instrument Cluster System [20], Elevator System [30], Nighttime Bank Deposit System [4], and Supermarket Checkout System [29].

Textual use case execution semantics defines how and when the various constructs of use cases should produce program behavior.

B. New Definition of Use Case Metamodel

Before presenting a new definition of a use case metamodel, we introduce the following concepts.

External Use Cases: The use cases that have S-SuD as their SuDs.

System Use Case Set: The set of all the use cases, including external use cases and those describing internal subsystems.

SuD Class: It has one attribute, sudName, used to record the name of SuD, and one SuD set of subsystems, denoted as subs.

User Class: This class has one attribute: userName.

Use Case Class: This class describes the information of a textual use case with the following attributes:

- uName: the name.
- sName: the SuD name.
- pActor: the primary actor.
- scope: the scope.
- uActivitySet: a string describing the activity sequence in this use case, and the string is formed by activities linked by operators: ‘;’ (sequence), ‘$or$’ (choice), and b∗() (loop).
- activities: the set of activities contained in the use case, where every activity is described by Activity Class.
- preCondition and postCondition specify the pre- and post-conditions of the use case, respectively.

Activity Class: This class has the following attributes:

- aName: the name of the activity.
- cType: the communication type of the activity: rendezvous communication.
- sender: the side that sends message out.
- receiver: the side that receives the message.
- condition: the condition under which the activity can occur.
- isUCReference: a label that indicates whether the description is a normal activity or refers to another use case.

Definition 2.2: A Use Case Metamodel is a structural description of the System Use Case Set with:

- USCModel: it contains name ‘sysName’ of the model.
- SuD: it contains 1, . . . , n SuDs, recording all the SuDs that occur in the System Use Case Set. Each SuD is an instance of SuD Class.

Fig. 3 displays the use case metamodel with UML diagram. Note that Scope is not included in the Use Case Metamodel. However, the information of Scope is reflected in the relationships between SuDs.

The Use Case Metamodel will be used as the source metamodel for the model transformation. By using ADT, it can be represented by an .ecore document. The adoption of a metamodel is justified as follows.

Our metamodel covers the basic model units and the unit relations. A desired metamodel should contain the following model units: Activity, Subject, Goal, Constraint, Connector, State and unit relations: Dependency, Generalization, and Association. In our metamodel, only Activity, Constraint, Connector, State, and Association are used.

Our metamodel is the first such model that contains (1) the subsystems and the relations among subsystems, i.e., the description of SuD, and the relation descriptions among SuDs; (2) the relations among all kinds of activities such as sequential, selection, and loop; and (3) the communication type by cType and the message direction by a sender and receiver.

III. Behavior Metamodel

The system to be modeled by textual use cases is a set of processes that interact with each other through a rendezvous communication mechanism. Thus, when it is implemented in the high level, we can map the system model to rendezvous-based languages such as CSP and Ada. We use a subclass of Petri nets to express formal textual use case execution semantics that allows program behavior to be checked, namely Message Passing nets, or MP nets for short. They are in fact
modified Ada-nets [28]. Instead of designing one par-begin-transition to start all processes and one par-end-transition to end all the processes, we design them such that each process has its own cycle. Such modification does not alter the behavior of the original programs but provides more proper design for a distributed program since different processes may have different speeds in a distributed environment.

A. MP Net

In what follows, the basic notations of Petri nets come from [39]. \( x^* \) is the set of inputs of \( x \), and \( x^\circ \) is the set of outputs of \( x \).

Definition 3.1: A Branch Net is a tuple \( P N^B = (P, T, F) \), where
- \( P \) is a set of finite Internal places, for each \( p \in P \), \( |^p| \geq 1 \), and \( |p^*| \geq 1 \);
- \( T \) is a set of finite Internal transitions, for each \( t \in T \), \( |^t| = 1 \), and \( |t^*| = 1 \); and
- \( F \) is the set of arcs such that \( F \subseteq (P \times T) \cup (T \times P) \).

Definition 3.2: A Closed Process Net is a tuple \( P N^C = (P \cup \{p_e, p_c\}, T \cup \{t_e\}, F) \), where
- \( P \) is a set of finite Internal places, for each \( p \in P \), \( |^p| \geq 1 \), and \( |p^*| \geq 1 \);
- \( T \) is a set of finite Internal transitions, for each \( t \in T \), \( |^t| = 1 \), and \( |t^*| = 1 \);
- \( p_e \) is a start place, \( |^p_e| = 1 \), and \( |p_e^*| \geq 1 \);
- \( p_c \) is an end place, \( |p_c^*| \geq 1 \), and \( |p_c| = 1 \);
- \( t_e \) = \{\( p_e \)\}, \( t_e^* \) = \{\( p_e \)\}; and
- \( F \) is the set of arcs such that \( F \subseteq (P \times T) \cup (T \times P) \).

It is easy to see that start place \( p_e \) of a closed process net leads one or many branch nets that end at \( p_e \) as shown in Fig. 4.

A Closed Process Net behaves as a state machine, and can be used to model a process.

The communications among processes are through rendezvous communication, which rely on mutual exclusion places \( p_{mcp} \) to the net as introduced in [28]. Such place guarantees that only one closed process net may send a token at any time to the receiving closed process net. No other closed process nets will send a token to that receiver until the current sender is acknowledged by \( p_{ack} \). Places \( p_{send} \) and \( p_{ack} \) are called buffer places, and the connected transitions are called communication transitions, as shown in Fig. 5.

Definition 3.3: A Rendezvous Communication Mechanism is a tuple \( CM^Q = (t_1, t_2, t_3, t_4, p_{12}, p_{34}, p_{send}, p_{ack}, p_{mcp}) \), where
- \( p_{12}, p_{34}, p_{send}, p_{ack} \) are places such that \( |^p_{12}| = |p_{34}^*| = 1 \), \( |p_{send}| = |p_{ack}| = 1 \), and \( p_{mcp} \) is a place such that \( |p_{mcp}| \geq 1 \).
- \( t_1 \) is a transition such that \( |^t_1| = |t_1^*| = 2 \), \( p_{12} = t_1 \), \( p_{send} = t_1 \), and \( t_1 \in p_{mcp} \); \( t_2 \) is a transition such that \( |^t_2| = |t_2^*| = 2 \), \( p_{34} = t_2 \), \( p_{ack} = t_2 \), and \( t_2 \in p_{mcp} \).
- \( t_3 \) is a transition such that \( |^t_3| = 2 \), \( |t_3^*| = 1 \), \( p_{34} = t_3 \) and \( p_{send} = t_3 \); \( t_4 \) is a transition such that \( |t_4| = 1 \), \( |t_4^*| = 2 \), \( p_{34} = t_4 \), and \( p_{ack} = t_4 \).

\( p_{12} \) and \( p_{34} \) are called intermediate states, \( p_{send} \) and \( p_{ack} \) are called Buffer places, and \( p_{mcp} \) is called a Control place, \( t_1 \) and \( t_4 \) are called output communication transition, and \( t_2 \) and \( t_3 \) are called input communication transition.

Definition 3.4: Let \( PN^C_1 \) and \( PN^C_2 \) be two closed process nets. If there exists a rendezvous communication mechanism \( CM^Q = (t_1, t_2, t_3, t_4, p_{12}, p_{34}, p_{send}, p_{ack}, p_{mcp}) \), such that \( PN^C_1 \) and \( CM^Q \) have common parts \( \{t_1, t_2, p_{12}\} \), and \( PN^C_2 \) and \( CM^Q \) have common parts \( \{t_3, t_4, p_{34}\} \), then \( PN^C_1 \) is Communicating with \( PN^C_2 \) through communication mechanism \( CM^Q \), or simply Communicating.

This definition describes that two closed process nets communicate with each other through a Rendezvous Communication Mechanism. If more closed process nets are communicating to the same other closed process net, then Rendezvous Communication Mechanism guarantees that only one closed process net may send a token at any time to the receiving closed process net. No other closed process nets will send a token to that receiver until the current sender is acknowledged.

Definition 3.5: Let \( A, B \) and \( C \) be three closed process nets. \( A \) and \( B \) are said to communicate with \( C \) at the same time, if their rendezvous communication mechanisms have the same place set \( \{p_{send}, p_{ack}, p_{mcp}\} \).
For example, Fig. 6 shows that two closed process nets $A$ and $B$ mutually send messages to the same closed process net $C$.

Our design can be extended to the case that one process sends requests to more than one process capable of servicing the requests and waits for at least one response as shown in Fig. 7. No additional detail about it will be given due to space limitation.

Definition 3.6: A Message Passing (MP) net is a subclass of Petri net $(P, T, F)$, consisting of a set of closed process nets $\{PN_i^C, i = 1, 2, \ldots, k\}$ and a set of Rendezvous communication mechanisms $\{CM_j^Q, j = 1, 2, \ldots, l\}$ such that

1) Each closed process net $PN_i^C$ is communicating through at least one communication mechanism; and
2) Each communication mechanism $CM_j^Q$ has been used for one and only one pair of process nets.

An MP net is denoted by $PN^M$. In it places are classified as idle, denoted as $PS = \{p_s\}$, buffer, denoted as $PB = \{p_{send}, p_{ack}\}$, control, denoted as $PC = \{p_{mcp}\}$, and activity for the rest, denoted as $PA$; while transitions are classified as activity, denoted as $TA = \{t||t| = 1, |t^*| = 1\}$, input communication, denoted as $TI = \{t\exists p \in PB, \text{s.t. } p \in t^*\}$ and output communication, denoted as $TO = \{t\exists p \in PB, \text{s.t. } p \in t^*\}$.

Thus, in an MP net, we can say that a closed process net has input and output communication transitions.

Definition 3.7: A marked MP net is a tuple $(PN^M, M_0)$, where

1) $PN^M = (P, T, F)$ is an MP net.
2) $M_0 : P \rightarrow \{0, 1\}$ is the initial marking: $\forall p \in PS, M_0(p) = 1$; $\forall p \in PA, M_0(p) = 0$; $\forall p \in PB, M_0(p) = 0$; and $\forall p \in PC, M_0(p) = 1$.

The obtained MP-net is bounded. In fact, we have the following result.

Theorem 3.8: A marked MP net $(PN^M, M_0)$ is 1-safe, i.e., $M(p) \leq 1$ for all places in $PN^M$ and for any marking $M$ reachable from $M_0$.

The proof can be found in Part I of Supplementary File.

From its construction we see that the proposed MP-net can be used to model a system consisting of a set of processes that communicate through Rendezvous Communication Mechanisms. The marking defines the state of a MP-net, or more precisely the state of the system described by the MP-net. The evolution of a state thus corresponds to an evolution of a marking.

The state changes are carried out by firing enabled transitions like an ordinary net [27]: a transition is enabled when all its input places have at least one token. When an enabled transition $t$ is fired, a token is removed from each input place of $t$, and a token is added to each output place of $t$; this gives a new state (i.e., a new marking).

The reachability graph of a Petri net captures the dynamic behavior of the net. The root node of the graph represents the initial state (i.e., the initial marking) of the net; directed arcs represent transition firings; and terminal nodes represent dead markings. The basic algorithm for the construction of the reachability graph is straightforward [27]. Each graph node is defined by a unique marking that corresponds to a reachable state.

The correctness property can be established based on the reachability analysis. Other semantics like [14], [25] to be used to prove the correctness will be developed in the future.

B. Format of Behavior Metamodel

The behavior metamodel is defined based on an MP net model, which is represented in a UML class diagram. This model has four classes: Net, Process, Place, and Transition.

Class Net: This class describes MP nets in general with

- $nName$: the name of an MP net,
- $Ps$: the closed process nets of an MP net,
- $cStates$: the buffer places for communication and control places.

Class Process: This class describes closed process net of an MP net with

- $pName$: the name,
- $sState$: the starting place,
- $eState$: the ending place,
- $eTransition$: the ending transition,
Fig. 8. Behavior Metamodel in a UML diagram.

— $S_s$: the set of internal places,
— $T_s$: the set of transitions.

Class Place: This class describes the places in an MP net with one attribute: $s\text{Name}$.
Class Transition: This class describes the transitions in an MP net with three attributes:
— $t\text{Name}$: the name of the transition,
— $preS(t)$: the set of input places of the transition,
— $nextS(t)$: the set of output places of the transition,
— $condition$: the condition for the transition.

The Behavior Metamodel is described in an UML diagram as shown in Fig. 8. By using ADT, it can be represented by an .ecore document.

IV. BUILDING SOURCE MODEL

Our source model conforms to the source metamodel and is generated by configuring the attributes in the metamodel. The required values for the configuration can be obtained from the set of use cases. The rest of this section shows how to get them.

Step 1: Building an Activity Table. In order to automatically get the attribute values, we need to extract all kinds of activity data from the sentences in the steps of each use case to build an activity table. The table contains a header part and a body part:

<table>
<thead>
<tr>
<th>No.</th>
<th>Activity-Name</th>
<th>Comm-Type</th>
<th>Sender</th>
<th>Receiver</th>
<th>Scope</th>
</tr>
</thead>
</table>

1) The header has $Use\ case$, $SuD$, $Primary\ Actor$, and $Scope$ whose descriptions are the same as those in the original use case.
2) The body contains:
   a) $No.$: It is the serial number which is the same as that in the original sentence.
   b) $Activity-Name$: There are three cases. (i) For the normal activities, i.e., they are not $abort$ or $go\ to\ step\ i$, we select the first verb (except $ask$) in the sentence and the first noun after the verb to form the $activity-name$, denoted as $activityName$. (ii) For the special activities $abort$ or $go\ to\ step\ i$, the $activity-name$ is $abort$ or $step\ i$, respectively. Especially, if an activity refers to another use case, that is, the name of the included use case is the activity description, then the $Activity-Name$ is set to ‘UC#No’, where No is the number of the included use case. (iii) For verb ‘ask’, if the first verb in the sentence is ‘ask’, we select the second verb and the first noun after it to form the $activity-name$.
   c) $Comm-Type$: If the activity is an internal activity, then the $comm-type$ is $null$; otherwise, $rendezvous$.
   d) $Sender/Receiver$: We have two cases. (i) The subject of the sentence is ‘System’, then the $Sender$ is the $SuD$ of the use case; For the $Receiver$, if the sentence contains an object information, then the $Receiver$ is the object, otherwise the $Receiver$ is null (an internal activity). (ii) The subject of the sentence is not ‘System’, then the $Sender$ is the subject, and the $Receiver$ is the $SuD$ of the use case. Note that we may also imitate the work [20] to create a stack to get the $Sender$ and $Receiver$.
   e) $Condition$: It is the pre-condition that is responsible for an activity to happen. In general, the pre-condition for activities in extension and variation are explicitly declared. For example,

Extensions:

3a The call is not connected.
3a1 System provides the failure information to the Driver.

Here the $Condition$ for the activity specified in ‘3a1’ is $cond = \text{‘The call is not connected.’}$ The activities in $extensions$ and $variations$ describe additional cases to the corresponding activity steps in $main$ scenarios, and these activities occur only when their conditions are satisfied.

StanfordParser$^2$ is employed to perform sentence analysis and obtain the context structure of each sentence. The Stanford Parser is a statistical natural language parser from the Stanford Natural Language Processing Group. The parser can read various forms of plain text input and can output various analysis formats. Most of the time, the probabilistic context-free grammar (PCFG) parser will be used.

Fig. 9 shows the PCFG structure generated by StanfordParser for the following sentence:

1 Seller submits item description.
This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Fig. 9. The PCFG structure tree of a sentence.

This structure then outputs a text file. The following is the output of the above structure:

```
(ROOT
 (S
  (NP (CD 1) (NN Seller))
  (VP (VBZ submits)
    (NP (NN item) (NN description)))
  (...)))
```

Note that StanfordParser can treat differently the “past” and “present” tense sentence styles.

Based on the output text file, we extract the activity data from the activity sentences via Algorithm 1.

**Algorithm 1: Building Activity Table**

**Input:** Textual Use Cases

**Output:** Activity Table

```
begin
  uc=readTextFile();
  uc=processedByParser(uc);
  pos=0;
  uc=getOneUC(uc,pos);
  while (uc)
    process(uc,lines);
    for each line (lines)
      if the sentence in (main scenario || variation || extension)
        if lines contains ’abort’
          str=getNumber(lines)+” abort”
        else if lines contains ’go to step’
          str=getNumber(lines)+” step”;
        else
          str=getSentence(lines);
          str+="."
          output str to output file.
        pos=getNextPos(uc,pos);
        uc=getOneUC(uc,pos);
      end
end
```

In Algorithm 1, function `processedByParser` processes the text with Stanford parser, and the output is the tagged text, function `getOneUc` fetches one use case from the text with `pos` as the beginning position, function `process` stores the analysis result of each sentence in one line, function `getNumber` gets the serial number of the sentence, and function `getSentence` extracts sentences from tagged text, and function `getNextPos` obtains the beginning position for the next use case.

Thus the activity data for the sentence in the above example are: `No. = 1, Activity-Name = submitsItem, Comm-Type = rendezvous, Sender = Seller, Receiver = System`.

After applying Algorithm 1, we can build the Activity Table for each use case.

**Step 2:** Obtaining Attribute Values. The following are the details to get all the attribute values from Activity Tables:

- **USCModel.** `sysName` is the system name.
- **SuD.** First, we obtain all SuDs. Going through all the Activity Tables, for each table, take the name of SuD. If SuD is not in the SuD set, add the SuD to it; otherwise, continue. Secondly, for each SuD, find the SuD sets for its subsystems. These ‘subs’ can be obtained based on the facts: (1) a use case’s SuD could be another use case’s `scope`; and (2) for a given use case, its SuD is one of the subsystems of its `Scope`. Thus, for two use cases `x` and `y` such that `x` has SuD `x_sud` and `scope x_scx`, and `y` has SuD `y_sud` and `scope y_scx`, if `x_scx = y_sud`, then `x_sud` is a sub-SuD of `y_sud`. This relation can be represented as:

```
⟨SUDs sudName = “y_sud”⟩
⟨subs sudName = “x_sud”⟩
⟨/SUDS⟩
```

In this way, we can find all the sub-SuDs for a given SuD.

- **User.** A user has only one item description, i.e. `userName`. All the primary actors in the Activity Tables built from external use cases constitute the User set.

- **Usecase.** We extract the following information from Activity Tables to build the Use Case classes: `uName`, which is the use case name; `sName`, which is the name of SuD; `pActor`, which is the primary actor; `uActivitySet`, which is a string describing the activity sequence; and `activities`, which is the set of all activities in the use case. Two steps are involved to construct `uActivitySet`:

  i) Obtaining the activities and the related information.

  The activity set can be constructed based on `activity-name, comm-type, sender, and receiver` in the body part of an Activity Table. For the special activities such as `abort` and the internal activities, no activity information can be extracted.

  ii) Obtaining the activity sequence. It contains all the activities of the use case, where the activities are connected by some operational symbols, e.g.,

  - ‘;’. Representing that two activities are in sequential relation in Main scenario. For example, if two activities `A` and `B` are in a sequential relation, then we have `A; B`.
  - ‘$’. Representing that two activities are in a selection relation, where one activity is in Main Scenario, and the other one is in Variation or...
exceptions in Extension. For example, if an activity $A$ has its alternative one $B$ in Variation, then we have $A \loror B$.

— $b^i()$. If under some condition $b$, the $i$th step (activity-$i$) is back to the $x$th step (activity-$x$), then all the activities between them are in a loop relation. For example, the third step activity $C$ is back to the first step $A$ under condition $b$, then we have $b^3(A; B; C)$, where $B$ is the second step activity. For example, the expression $BE_1; A; (BE_2 \loror abort)$ means that activity $A$ has an corresponding activity abort in Extension. $BE_1$ represents the system activities before $A$, and $BE_2$ represents the system activities after $A$. Similarly, $BE_2; (A; BE_2) \loror abort$ indicates that $A$ has an exceptional activity abort in Variation.

We have Algorithm 2 to build a use case source model from Activity Tables.

**Algorithm 2: From Activity Table To Use Case Model**

**Input:** Activity Table

**Output:** Use Case Model specified in xmi

begin
text=readFromTable();
outputHeaderOfUSCModel(text);
outputUoDRelations(text);
getUsers();
pos=0;
puc=能得到UC(text,pos);
while (puc)
  uName=能得到UName();
sName=能得到SName();
pActor=能得到PActor();
output the header information of each use case;
str1="","str2="-"+uActivitySet=";
for each activity description $ti$
  aName=能得到AName($ti$);
  cType=能得到CType($ti$);
  sender=能得到Sender($ti$);
  receiver=能得到Receiver($ti$);
  act="$ti$ has an corresponding activity abort in Extension. $BE_1$ represents the system activities before $A$, and $BE_2$ represents the system activities after $A$. Similarly, $BE_2; (A; BE_2) \loror abort$ indicates that $A$ has an exceptional activity abort in Variation.

We have Algorithm 2 to build a use case source model from Activity Tables.

In Algorithm 2, function getOneUC fetches one row from a given Activity Table, function getName gets the use case name, other ‘get’ functions have the similar explanations. str2 collects the value for uActivitySet. Hence, based on the Step 1-2, we can construct a use case source model.

V. FROM SOURCE MODEL TO TARGET MODEL

The Target Model can be obtained by configuring the target metamodel with the values obtained from the model transformation. The model transformation is through executing a set of rules written in ATL (Atlas Transformation Language) in the platform provided by Eclipse. ATL is a model transformation language in the field of Model Driven Engineering (MDE). We can produce a set of target models from a set of source models using an ATL program. Eclipse provides an Integrated Development Environment (IDE) for developing ATL programs. To perform model transformation, the transformation rules should be designed first, and then implemented in ATL language. ATL programs must guarantee that the implementation is consistent with the transformation rules. It is the rules that determine what properties can be preserved. The domain of these rules is the attribute values in the Source model and the image is the values used for the target configuration.

Next, we present model transformation rules. In general, one USCModel is transformed to one Net, every SuD (except S-SuD) or User in USCModel is transformed to a Closed Process Net in the behavior model, and an activity is transformed to a Transition. Besides, an activity also generates an outgoing internal place to the transition, which represents the new state caused by the activity. A use case corresponds to a branch net in the closed process net. The following notations are given before presenting the rules.

- The names of the transitions and places in a SuD-based process are $\text{sudName}+::+\text{aName}$. For example, ‘$CS'$-locateOffer could be a transition or a place, where ‘$CS'$ is a $\text{sudName}$ and ‘locateOffer’ is an $\text{aName}$. If it is a transition, then it belongs to the set $T$: if it is a place, then it belongs to the set $S$.

- The names of the transitions and places in a User-based process are $\text{userName}+::+\text{aName}$. For example, ‘$U'$-submitItem means that transition submitItem is in a User-based process.

- The names of the Buffer places are ‘$B$-+$\text{aName}$.

- The names of the control places are ‘$C$-+$\text{aName}$.

A. Net (nName, Ps, commStates)

The rules to transform USCModel to a net are defined as follows.

- **nName**: The name of the Net is the name of USCModel.
- **Ps**: The closed process nets can be obtained as follows: (i) A SuD (except S-SuD) from USCModel is mapped to a closed process net whose name is the name of the SuD; and (ii) A User in USCModel is mapped to a closed process net whose name is the name of the User.
- **commStates**: There are two kinds of $\text{commStates}$: Buffer place (‘$B$-+$\text{aName}$) and Control place (‘$C$-+$\text{aName}$). An activity that describes the interaction between different subsystems is mapped to a buffer place. If multiple processes send messages to one process through one buffer place or one process sends messages to multiple processes through one buffer place, then a control place is required.
B. Process (pName, sState, eState, eTransition, Ts, Ss)

1) Process Generated From SuD: Here SuD is not S-SuD. Each SuD related use case can generate a branch net in the closed process net. Since each SuD is triggered by a user action, and interacts with other SuDs to complete a service, the first transition of each branch net is an Input Communication transition. The process structure can be designed as one sState place followed by several branches, where a branch is executed if its first transition is enabled and fires. The corresponding Petri net structure is sketched in Fig. 10.

The transformation rules for a closed process net from SuD are defined as follows.

- **pName**: The name of the closed process net is the name of SuD, i.e., sudName.
- **sState, eState** and **eTransition**: Every closed process net has a start place (sState), an end place (eState), and an end transition (eTransition). eTransition has eState as its input place, and has sState as its output place.
- **Ts**: It is a set of transitions generated from activities. At the moment these transitions have the names only, which are made of SuD name and activity name: sudName+anName+T, while the other information will be obtained later when we define transformation rules for Transitions.
- **Ss**: It is a set of internal places generated by activities: each place is an output to the transition generated from an activity, and the name is sudName+anName+P.
- **The integration of branch nets.** The branch nets can be integrated based on the preCondition and postConditions of their corresponding use cases. The branch nets can be integrated based on the preCondition and postConditions of their corresponding use cases. Let br_i, br_j, and br_k be three branch nets, with their corresponding use cases as U_i, U_j, and U_k, respectively. The principle of their integration is described as follows. (1) For br_i and br_j, if U_i.SuD = U_j.SuD and U_i.postCondition = U_j.preCondition, then they can be merged into a new branch net br_new with a sequential order: we first rename br_j.sState as ‘connectij,’ then replace br_j.eState in br_i.lastTransition.nextS with br_j.sState, and finally, br_new = br_i.br_j. (2) For br_i, br_j, and br_k, if U_j.SuD = U_j.SuD = U_k.SuD and U_j.preCondition = U_k.postCondition & & cond and U_k.preCondition = U_k.postCondition & & ‘cond, where cond refers to a condition description (whenever the preConditions of U_j or U_k are satisfied, U_i.postCondition is also satisfied), then br_i, br_j and br_k are integrated into a new branch net br_new with a selection structure: we reset the name of br_i.eState as ‘connectijk,’ then replace the sState in br_j.firstTransition.preS and br_k.firstTransition.preS with br_i.eState, and finally, br_new = br_i; (br_j $or$ br_k).

2) Process Generated From User: In this situation, SuD is S-SuD. One User can generate a closed process net. The User initiates an execution by passing information to the system. Thus, the first transition of each branch net is an Output Communication transition. The process structure can be designed as one sState place followed by several branches, which is sketched in Fig. 11. Every branch net represents a service required (or obtained) by the user. The branches will be executed based on the user selection.

The rules to get pName, sState, eState, eTransition, Ts, and Ss are the same as those for closed process nets from SuD. Note that the use cases here are external use cases.

C. Transition (tName, preS, nextS)

We have two types of transitions: one generated from SuD and the other generated from User.

1) Transitions From SuDs: Here SuD is not S-SuD. If an activity does not interact with other use cases, then this activity generates an internal transition; if it belongs to a Sender, then it generates an output communication transition; and if it belongs to a Receiver, then it generates an input communication transition. The basic rules are as follows:

- **tName.** The tName of the transition is the anName of its corresponding activity. It is an element of Ts obtained in the Process.
**nextS(t).** For a transition t, if it is an internal one, then t* contains one internal place; if it is an input communication one, then t* contains an internal place and a buffer place; if t is an output communication, then t* contains an internal place and a buffer place. If t is generated from the last activity of the use case, then t* contains eState of the process. To compute a transition’s nextS, its corresponding activity and the uActivitySet of its corresponding use case are used. On one hand, the type of the activity determines the number of places in nextS; On the other hand, after locating the name of the current activity in uActivitySet, the next activity can be identified. Note that, the nextS of the last transition of a branch contains eState instead of an internal place.

• **preS(t).** For a transition t, if it is an internal one, then t* contains one internal place; if it is an input communication or an output communication one, then t* contains an internal place and a buffer one. If t is generated by the first activity of the use case, then t* is sState of the process. The computation of preS is similar to that of nextS except that preS of the first transition of a branch net contains sState instead of an internal place.

• **condition.** The condition of the transition is the condition of its corresponding activity.

Note that, if an activity ‘a’ refers to another use case ‘#no’ (the serial number of the use case), then the name of the activity is a#no. The name of its generated transition also has ‘#no’ as that of the regular activity. However, such transition represents a branch net from use case ‘#no’, and this branch net can be integrated into the current branch net.

2) **Transitions From Users:** The method to get the transitions of the closed process nets from User is quite similar to that from Suds except:

i) In 1) of Section V-C, the activities come from the use cases whose SuDs are not S-SuD, while here they come from the external use cases whose SuD is S-SuD. Only user related activities can generate transitions.

ii) **preS and nextS rely on the uActivitySet of the use case.** In 1) of Section V-C, the activities in uActivitySet are SuD related, while here for a User-related use case, they are not necessarily User-related. Thus, we need to filter uActivitySet such that all the activities left are User-related. Then we can use them to generate preS and nextS. The filtering process is designed as follows.

Suppose that the uActivitySet of a use case (a S-SuD related use case) has the form: \( (a_1; a_2; \ldots; (a_i \text{or} \text{sor} \text{skip}); \ldots; a_j; (\text{skip} \text{or} \text{sor} \text{abort}); \ldots; a_n) \), where all \( a \)'s are activities. For each activity in the uActivitySet, the decision to keep or delete it is made based on whether it is a user-related activity or not. (1) An activity has no relevant activities in extension or variation. For example, consider the first activity \( a_1 \) in uActivitySet, if it is a user-related activity such as \( a_1 \text{sender} = \text{userName} \) or \( a_1 \text{receiver} = \text{userName} \), then keep \( a_1 \), and otherwise, delete it, uActivitySet becomes \( (a_2; \ldots; (a_i \text{or} \text{sor} \text{skip}); \ldots; a_j; (\text{skip} \text{or} \text{sor} \text{abort}); \ldots; a_n) \). (2) An activity has a relevant activity in extension or variation. For example, consider \( a_i \) and \( a_j \). If \( a_i \) is a user-related activity, then keep \( (a_i \text{or} \text{sor} \text{skip}) \) in uActivitySet, and otherwise, delete it. Similarly, if \( a_j \) is a user related activity, then keep it, and otherwise, delete it.

iii) **For the first transition of each branch, its preS is sState of the process, and its nextS contains an internal place, and an output buffer place.** This is different from that in 1) of Section V-C, where preS of the first transition contains sState and an input buffer place, and the nextS contains an internal place and may contain an output buffer place.

The proof of correctness of the model transformation can be found in Part III of Supplementary File.

VI. CASE STUDY

A. Description of the Prototype

After the model transformation supported by Eclipse, we obtain a configured Petri net model which is written in *xmi*. Then a Java program reads all the data from this *xmi* file and displays a Petri net. We have developed a prototype3 to support the model transformation. It contains the following four parts and can be plugged into Eclipse platform.

• **The editor of textual use cases.** The editor allows one to open a predesigned use case or add a new created use case into the current use cases set. In this part, all textual use cases are checked including sentence grammar and style.

• **The builder.** The builder generates activity tables from use cases and then generates a use cases model from activity tables. In this part, StanfordParser is integrated to analyze the sentences of the use cases.

• **The converter.** The converter transforms a source model to a target one, and is implemented by nineteen ATL helpers.

• **A Petri net display.** Both sketch and detailed Petri Net models of use cases are displayed.

Comparing with such commercial tools as ‘Ravenflow’ (www.ravenflow.com), which generates model for documentation, our tool can generate a model that can be directly used for verification and simulation.

B. An Example

In order to illustrate the proposed method, the Functional Requirements Specification of European Integrated Railway Radio Enhanced Network (EIRENE)4 is employed as an example. An EIRENE network is a railway telecommunications network, based on the ETSI GSM standard. The EIRENE Functional Requirements Specification defines the requirements of a radio system satisfying the mobile communications needs of the European railways. It encompasses ground-train voice and data communications, together with the ground-based mobile communications needs of track side workers, station and depot staff and railway administrative and managerial personnel.

3https://sourceforge.net/projects/texturalhc2pnu/
Cab Radio is a part of the specification, covering driver call related functions, other driver-related functions, other cab radio functions, environmental and physical requirements, driver man-machine interface, driver safety device interface, train-borne recorder, control/command interfaces, and other interfaces. We deal with only the function Send Railway Emergency Call in the part of driver call related functions in this paper due to space limitation. This function involves three sub systems: Cab Radio, Man-Machine Interface (MMI), and Driver. Fig. 12 outlines their relations. The scenario is as follows. To make a railway emergency call, the Driver needs to send (receive) information (indication) through an MMI-based device. Firstly, the Driver initiates a call through MMI. The call information is passed to Cab Radio, and meanwhile the call will also be sent to train-borne recorder that is connected to the Cab Radio. The Cab Radio then connects the call after it receives the connected request. An indication is sent to the MMI from which Driver gets the connection indication. The call is connected to the Loudspeaker until the Driver picks up the Handset. Finally, the Driver terminates the call when the call is finished, and the Cab Radio will send an indication after the termination signal is received, and this kind of indication is passed to the Driver through MMI. The semantics of the scenario can be described by UML sequence diagram as shown in Fig. 13.

Fig. 12 provides guidelines for writing use cases. Each part described in it can be considered individually and use cases can be designed for it. We will use three use cases shown in Fig. 14 to illustrate our model transformation process.

Based on the use cases provided in Fig. 14, we can construct their corresponding Petri net model for the EIRENE system with the help of the prototype tool (see Part IV of Supplementary File).

Fig. 15 shows the part of the model that contains three use cases only. Via our prototype tool, we can get the MP net model of the entire EIRENE system.

The generated Petri net model well models system behavior such as concurrency. It can not only be used for verification, but also for simulation. Since Petri nets have graphical notations, if the checking tool detects errors, then they can be displayed in the picture. Thus we can easily find the problems in the use cases.

C. Requirement Checking

Now the requirement has its formal presentation, and hence we can check its properties. We can check them with a Petri net tool INA³ and a model checking tool SPIN [34]. Based on the results, we can locate the errors or possible errors. For model checking, we need to map the Petri net to Promela presentation, and then apply the model checking technique. The mapping details can be seen in [10]. Let us check the completeness, consistence and correctness.

- Completeness: This means that the requirements are complete. Reflecting in our Petri net, (1) there are no isolated subnets, and (2) every place must have both input and output transitions.

Checking Net. INA can automatically perform the analysis of a net to check structural properties. For (1), it will display ‘The net is connected’; for (2) it will display ‘The net has places without pre-transitions/post-transitions.’

³http://www2.informatik.hu-berlin.de/lehre/studium/automaten/ina/
Locating Errors. For an isolated net, we can use the places and transition in the isolated net to locate the corresponding use cases to determine missed parts. For a place without input/output transition, the corresponding use case requires messages from other use cases, or has to send messages to others.

- Consistence: This means that two or more requirements are not in conflict with one another. It usually includes many aspects [15] in the model. In our case, we consider the consistence in both syntax and semantics levels. For the former, (1) the net of a use case $U$ is consistent with that of $U$’s included use case, i.e., the name of included use cases should exist in the transition names of the net model for use case $U$; (2) For two process nets, if they communicate, then the names of the input transition and output transition should match. For the latter, every place is reachable.

- Checking Net. For syntax checking, we may do net reviewing. Semantics property checking can be conducted via INA. INA can display messages such as ‘The following transitions are dead at the initial marking,’ ‘The net is not live,’ ‘The net has dead reachable states’ and ‘The net is not statically conflict-free.’

- Locating Errors. If a net has syntax errors, for (1), we locate use case $U$ and check every step of use case $U$. For (2), we check the use case that has the step corresponding to input transition/output transition. If deadlock exists, based on the dead transitions, we can analyze the corresponding activities to find the conflicts in the requirements. Model checking can tell where the deadlock is in the net model based on the location where the error trace stops, and the information can be used to trace the conflicts in the use cases.

- Correctness: It means that the requirement itself should be correct. Reflecting in the net model is that (1) the reachability correctness, i.e., the traces being correct, and (2) boundedness.

- Checking Net. For (1) INA can print the trace from the initial marking to the target marking; for (2) it can display ‘The net is bounded’ or ‘The net is not bounded,’ and even more ‘At least the following places are unbounded.’

- Locating Errors. If a trace is not correct, we may analyze if it has a missing event, extra event, or wrong order. Then locate the corresponding use cases based on the events, and find if there are some steps that are missed or extra or in wrong order in these use cases. For boundedness, there is an overflow in the net. Based on the unbounded places from INA, we can locate the error in the use cases.

Before entering the next development phase, we may map the generated Petri nets to Haskell, a functional language, to obtain a high level implementation of the system.
Haskell program, we can check in the early design phase such problems as race conditions [38].

VII. DISCUSSION OF RELATED WORK

Our research is closely tied to the following work.

Behavior Models from Textual Use Cases: Gutierrez et al. [13] have developed a process to automatically generate activity diagrams from textual use cases. The generation is performed by a model transformation that is defined in QVT-relational language. They also consider structural use cases. However, they use more restricted language style than ours to describe functional requirements. Besides, they use UML activity diagrams to model the system behavior, which is not easy to be checked. In our case, we use Petri net to model system behavior, which can be converted into the input languages of different checking tools [34], such as SPIN [17].

Barrett et al. [1] consider a use case model, namely, DSRG-style use case structure. It has well-defined formal syntax and semantics. Use cases can be transformed to the use case models that can be merged. Their technique focuses only on model merging and detecting merging inconsistencies and conflicts without considering the synthesis of behavioral models.

Wang et al. [36] present a semi-automatic approach for constructing feature models from use cases. After identifying similar operations, objects, and relationships among the objects by analyzing the set of the use cases, Application Feature Models can be derived from each use case. Then, they are checked and adjusted to eliminate improper features and relationships. After they are adjusted, they are automatically merged into a Domain Feature Model that represents partial characteristics of the domain. However, they fail to obtain the overall characteristics of the domain.

Gordon and Harel [12] present an initial natural language interface that generates live sequence charts (LSC) from structured English requirements. The structured English is much richer and much expressive than the our work’s, e.g., allowing to specify negative scenarios. Thus the synthesis from this scenario language is much more difficult [23].

Synthesizing Petri Net From Use Cases: Based on traditional P/T nets, by assigning a constraint to each transition, Lee et al. [22] propose a new type of Petri nets, called Constraints-based Modular Petri Nets (CMPNs) to formalize the informal aspects of use cases. CMPNs consist of a set of nets, each of which models an individual use case. However, it cannot be used to describe the interactions between subsystems. Besides, the conversion process from use cases in a natural language cannot be completely automated and interactions between users and domain experts are still needed. By contrast, our net model can be used to describe interactions between the subsystems through a rendezvous mechanism, and more importantly, our conversion process is automatic.

In [37], the use cases are described by use case cards, and the scenarios are constructed based on some restricted event flow format. These scenarios can be translated to a class of timed and controlled Petri nets by defining some rules. After analyzing the Petri nets, properties such as completeness, consistency and correctness can be checked.

Somé [32] proposes a formal control-flow based semantics for use cases. The formalization starts from a formal definition of use case syntax: a UML metamodel as abstract syntax and a restricted natural language as concrete syntax. The basic Petri net formalism is used to express the semantics, which are generated through 13 mapping rules. The algorithms are given to synthesize a two-level state model from a set of sequentially related use cases. However, their net model focuses on the interaction between individual use cases, and cannot give much design information. Besides, their mapping rule induces an exponential complexity.

In [3], use cases are formally specified as labeled Petri nets, namely, C-nets that are themselves live, safe and reversible. The nets are then synthesized into one single net that serves to represent the integrated system. Based on their synthesis methodology, the correctness, safeness and conservativeness of the resulting model can be achieved. A C-net is quite similar to CMPN [22] since both describe a single use case. By contrast, this work deals with multiple use cases described with natural languages, and each use case is mapped to a branch in the process net that describes a subsystem. Our synthesis rules are based on the definition of an MP net. Moreover, our target behavior model can be automatically generated by model transformation.

VIII. CONCLUSION

We have presented a method to automatically map textual use cases to Petri net based behavior models which was not seen before to the best knowledge of the authors. We have implemented our approach in a model-generation tool that is integrated with the ADT platform provided by Eclipse.

In the future work, we will address three important issues:

- Although we have 6 types of sentence structures for the use case writing, it is not powerful enough to represent all kinds of requirements, especially those requirements described by modal verbs and fuzzy words. Hence, more sentence structure types need to be added to the sentence base.
- From user textual requirements to use case models, we adopt language processing techniques. The accuracy of the use case models largely depends on such techniques. Although we can extract all the required values for the use case model, it is possible that we may lose some information that is not considered in the use case metamodel, such as performance and reliability. Accordingly, we need to continue to work on language processing techniques that will enable the generation of richer component models.
- Our proposed net model can model the concurrent systems whose communications among processes are through synchronized message passing. Even though, the model is still basic. For examples, it can not model unbounded number of processes like Erlang. It can not model concurrent systems with asynchronous message passing, which is another type of message passing such that the sending side does not wait for a response. Hence, our model cannot model all possible behaviors of systems communicating via message passing semantics.
• We plan to manually build a Petri net model for the whole Functional Requirements Specification of European Integrated Railway Radio Enhanced Network (EIRENEN), to validate our method. We believe that this is a hard work, which will further demonstrate the benefit of our method.

References