Towards Automatic Generation of Formal Scenarios Specifications from Real-Time Reactive Systems Requirements Written in NL

Las Vegas Nevada, USA, June 26-29, 2006

O. Ormandjieva, I. Hussain
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada

Abstract - This paper describes a new method for automatic generation of formal scenario-based specifications from requirements written in structured natural language (NL). The goal is to allow for completion of the formalization of the real-time reactive systems development process from users’ requirements. We believe this method would guarantee the correctness and the completeness of the scenarios generated in the requirements specification phase, thus decreasing the probability of errors in the subsequent development phases. To our knowledge, the approach is completely new in the field of requirements engineering.

Keywords: Real-Time Reactive System, Generation of System-Level Formal Scenarios, Requirements Engineering, Natural Language Processing.

1.0 Introduction

The correctness and completeness of the scenarios specifications generated in the analysis phase is crucial for the development of quality software, as errors introduced at this stage and propagated to implementation are extremely costly to fix. This is particularly true for real-time reactive systems [2] which are largely event-driven, interacting intensively and continuously with the environment through stimulus-response behavior and are regulated by strict timing constraints. Examples of such systems include emergency alarm systems, air traffic control systems, nuclear reactor control systems and telecommunication systems.

The goal of this paper is to introduce a new approach for automatic generation of formal scenario-based specifications from textual requirements of real-time reactive systems written in structured English. In the context of real-time reactive systems, which are mostly safety-critical, the main motivation for formalization of the development process comes from the requirements for correct implementation of time-dependent behavior. Our aim is to assist developers in the tedious task of developing correct and complete scenarios for such complex systems. To guarantee the correctness, we propose to automatically generate all possible traces of interactions between the users and the system (giving rise to scenarios), which exhaustively cover the possible behaviors of the system. The completeness is achieved through automatically selecting only a subset of scenarios, which represent the critical system's behavior, based on certain criteria.
The paper is organized as follows: section 2 introduces our methodology; the detailed algorithms are described in sections 3 to 5, followed by an illustration of our approach on a case study in section 6. The performance assessment is explained in section 7. The conclusions and directions for future work are outlined in section 8.

2.0 Our Methodology

The goal of this research is to reduce the user’s involvement in scenario-based specification and to allow for formalization of the development process right from the user's requirements. The importance of introducing formalization as early as the requirements phase is enormous. It guarantees that the problem is correctly understood throughout a complex development process, and that a high-degree of dependability of the software is achieved in a specific environment. Our systematic approach to interaction patterns mining from requirements (including the timing constraints) and to mapping them to scenarios specifications is explained below.

2.1 Approach

At the requirements phase, we view the system as a black box, and thus only the environmental events crossing the system's boundary are observable. We apply a scenario-based method to model the system's usage from the user's point of view. The scenarios describe, in a temporal sequence, the interaction between the system and its environment for achieving a usage goal, which is performed through interchanging environmental events (visible from outside the system) (see, for instance, [4]). The environmental events are of two types: Input (from the environmental object(s) to the system, including those that trigger the functionality of the system) and Output (from the system to the environmental object(s)).

The real-time reactive systems are characterized by their hard timing constraints. The traditional specification approach assumes that timing constraints are given as statements explicitly by the users, and then specified formally in some notation like temporal logic. Typically, the constraint on the response time is provided at the design phase, and the verifier automatically determines whether or not it is satisfied. These techniques do not provide feedback on the system's deviation from its expected performance. However, such information can be extremely useful in fine-tuning the behavior of the system. In our approach, the timing requirements are incorporated in the requirements specification phase as a global clock event, a "Tick" abstracting one time unit, and an "NTR" abstracting the "No Timing Requirements" statement. The incorporation of the Ticks into the patterns of interactions allows for performance analysis of the minimum/maximum delay time for each system usage during the design validation and testing phase, as described in this paper.

We extract the sequencing and the timing constraints from the requirements documentation in the form of rules, which relate: i) Input events to Input events; ii) Input events to Output events; iii) Output events to Output events, iv) Output events to Input events; and v) Trigger events to Last events, where "Last" means an event that finalizes the usage of the system triggered by the Trigger event. The above knowledge is sufficient to produce generic patterns of interactions between the environmental objects and the system. The first event in such an interaction pattern is always a triggering event from an
environmental object (i.e., Input event), and the order between the events in one pattern should satisfy the sequencing rules for partial order between the objects. We call such a pattern a legal sequence.

2.2 Format of the Requirements written in NL

We require the user requirements to be written in structured NL in order to facilitate their automatic analysis. We propose the following EBNF Grammar for the Requirements Description File (RDF) for the user requirements document:

\[
\begin{align*}
\text{<requirements_description_file>} & = \text{name_of_the_system}, \text{body}, \text{EOF} \\
\text{name_of_the_system} & = \text{letter}, \{\text{letter|digit}\} \\
\text{body} & = \{\text{NL}, \text{sentence}\} \\
\text{sentence} & = \text{statement}, \text{min-delay/max-delay} \\
\text{statement} & = \text{conditional} | \text{clause} \\
\text{conditional} & = \text{IF|WHEN|AFTER}, \text{clause}, \text{THEN|NOP|OR} \\
\text{clause} & = \text{statement}, \text{AND|OR}, \text{statement} \\
\text{sender/subject} & = \text{concept} \\
\text{message/action} & = \text{letter}, \{\text{letter|digit}\} \\
\text{recipient/object} & = \text{concept} \\
\text{min-delay/max-delay} & = \text{time}, \text{time} | \text{NTR} \\
\text{time} & = \{\text{digit}\} \\
\text{concept} & = \text{letter}, \{\text{letter|digit}\}
\end{align*}
\]

2.3 Method

In our approach, the requirements specification process consists of three main activities: i) generation of an exhaustive set of legal sequences from the textual requirements (see section 3, LSG Algorithm); ii) reduction of the set obtained in Step (i) to a subset of legal sequences called a canonical set (see section 4, CSG Algorithm), based on which the minimum/maximum delay of the performance measurements are to be later validated; and iii) mapping the canonical sequences to a scenario’s formal presentation which will serve as a basis for design modeling and performance validation (see section 5, GSLS Algorithm).

The algorithms for our requirements specification process are introduced below, followed by the Railroad Crossing case study which illustrates our methodology.

3.0 LSG Algorithm

The LSG algorithm generates the exhaustive set of legal sequences from a given set of requirements captured in an RDF format. The inputs to the LSG algorithm include the set of environmental events \(\text{EnvE}\) and the set of triggering events \(\text{TE} (\text{TE} \in \text{EnvE})\), both sets predefined by the user. If there are more than one instances of a given object, the events sent or received by each instance are indexed in order to distinguish them.
Step 1. Generation of Sequencing Rules

Step 1.1 Store the name of the System:
The <name_of_the_system> is one of the most integral parts of the RDF. It indicates the name of the system to be developed. Thus, the <name_of_the_system> is first stored in a temporary location by the algorithm.

Step 1.2 Identify the Inputs and the Outputs:
We then need to scan the whole text of the RDF to identify Inputs (messages coming into the system from outside the boundary) and Outputs (messages going out of the system, beyond the boundary). Note that the Inputs and Outputs belong to the user-defined set EnvE. Thus, each <statement> of the RDF is scanned as follows:
(1) For each first occurrence of <statement> do:
(2) If <name_of_the_system> is found in the <recipient/object> part of the <statement>
Then the <message/action> of the <statement> is added as “Input” to the set EnvE
Else
If <name_of_the_system> is found in the <sender/subject> part of the <statement>
Then the <message/action> of the <statement> is added as “Output” to the set EnvE

Step 1.3 Build a Directed Acyclical Graph:
Now, we build a directed acyclical graph (DAG) representing the relationships among the Inputs and Outputs. We first add vertices to the graph, as follows:
(1) For each Input or Output present in the list of messages, i.e. the set EnvE, do:
(2) Add a vertex to the graph indicating the name of the message/action and its type (i.e. Input or Output).
We then scan the RDF for the <conditionals>, since each <conditional> can represent one or more edges between vertices.
(1) For each <conditional> found do:
(2) For each <statement> found before “THEN” or “,” do:
(3) Select the vertex corresponding to the <statement> as the “start point”.
(4) For each <statement> found after “THEN” do:
(5) Create an edge starting from the “start point” and ending at the vertex corresponding to the <statement>.

Step 1.4 Topologically Sort the Graph
To obtain the primary sequence (PS) of Inputs and Outputs, we topologically sort the graph, as follows:
(1) Do while the graph has one or more vertices:
(2) Find a vertex with no incoming edge.
(3) Add the name of the message/action and its type (i.e. Input or Output), which is associated with the current vertex, to the primary sequence of Inputs and Outputs, i.e. PS.
(4) Delete the vertex.
(5) End while

Step 1.5 Generate the set of Sequencing Rules R
We exhaustively generate, from the primary sequence (PS) of Inputs and Outputs (obtained in Step 1.4), all possible Input-Input, Input-Output, Output-Input and Output-Output combinations, resulting in a set R of sequencing rules.
Step 1.6 Add Timing Requirements
For each sequencing rule R: if the events are constrained by <min-delay/max-delay> in the corresponding RDF statement, then replicate the rule to include all possible time intervals between <min-delay> and <max-delay> (i.e., <min-delay> number of “Tick”s, <min-delay>+1 “Tick”s, etc. until the <max-delay> number of “Tick”s is reached).

Step 2. Exhaustive Generation of Legal Sequences
Next, the LSG algorithm generates the set $\Sigma$ of all possible legal sequences of interactions between the environment and the system by actually combining all possible rules in a set R for all possible triggering events in the user-defined set TE.
We claim that the resulting set $\Sigma$ exhaustively covers all possible behavioral patterns of system-environment interactions, and it is correct because of the automatic approach to its construction. The set $\Sigma$ is a finite set, although a very large one.

4.0 CSG Algorithm

We reduce the set $\Sigma$ to a smaller canonical set C of pairs of legal sequences $(\sigma_{min}, \sigma_{max})$ covering all environment-system interaction patterns for both minimum $\sigma_{min}$ and maximum $\sigma_{max}$ delays. For instance, the canonical set of the railroad crossing problem with one Train and one Gate consists of only one pair of legal sequences.
The CSG algorithm makes use of the relation $\prec$ on the elements of the set $\Sigma$, such that, for two legal sequences $\sigma, \sigma' \in \Sigma$, we state that $\sigma \prec \sigma'$ if the sequence $\sigma$ is a subsequence of the sequence $\sigma'$. The relation $\prec$ is a partial order relation on the set $\Sigma$. We construct a set of pairs of sequences C, where each sequence belongs to the set $\Sigma$, as follows:
1. Initialize $C = \emptyset$;
2. Choose a legal sequence $\sigma \in \Sigma$, which is not yet “chosen”;
3. Construct the minimal sequence $\sigma_0 \prec \sigma_1, \ldots, \sigma$ and the maximal sequence $\sigma \prec \sigma_k, \ldots, \sigma_{m}$, where $\sigma_0$ is a subsequence of all sequences $\sigma_1, \ldots, \sigma$, and $\sigma_m$ is not a subsequence of any sequence in the set $\Sigma$;
4. Set $C = C \cup \{\sigma_0, \sigma_m\}$;
5. Mark all the scenarios chosen in Step 3 as “chosen”;
6. If all scenarios in $\Sigma$ are “chosen”, terminate; otherwise repeat from Step 2.
The set $\{\sigma_{max} \exists i \cdot (\sigma_i, \sigma_{max}) \in C\}$ contains sequences that are maximal in the sense that no sequence in it is a subsequence of another sequence in it, and, for every sequence in it, the set $\Sigma$ includes several subsequences. The set $\{\sigma_{min} \exists i \cdot (\sigma_{min}, \sigma_i) \in C\}$ contains sequences that are minimal in the sense that the sequence in it is a subsequence of a set of other sequences. We claim that the set C represents a canonical collection of legal sequences covering all functional requirements as specified by the user, therefore we can guaranty the completeness of the scenario-based specifications by construction. Moreover, the set C is sufficient to model the system's critical behavior, and to evaluate the performance of the system with respect to its functional...
requirements and the minimum/maximum delay. The generated canonical sequences are mapped to formal system-level scenarios, including the minimum/maximum delay time for each system usage, as described in the GSLS Algorithm.

5.0 GSLS Algorithm

In our work, we formally specify a system-level scenario $sls$ of a system's usage as a tuple:

$$sls = \{SE \cup \{Tick, NTR\}, MET, ET\}$$

where $SE$ represents all the environmental events participating in the scenario, $MET$ is a mapping from $SE$ to the time axis, and $ET$ specifies the expected time delay of system's execution for this scenario. The GSLS algorithm generates system-level scenarios from the set of canonical legal sequences. The input to the algorithm is the canonical set $C$, the legal sequence elements of which belong to the user set of environmental events $EnvE$. The output is a collection of formal specifications of system-level scenarios $SLS$. The algorithm is described below:

**Step 1.** Choose one non-marked pair of sequences $(\sigma_{\text{min}}, \sigma_{\text{max}}) \in C$, and mark it as "chosen". If all elements of $C$ are marked, terminate the algorithm.

**Step 2.** Add all environmental events participating in $\sigma$ to:
- the set $sls_{\text{max}}.SE$ for $\sigma_{\text{max}}$
- set $sls_{\text{min}}.SE$ for $\sigma_{\text{min}}$

**Step 3.** Create the set $sls_{\text{min}}.MET$ as follows:

(a) $e_i = \text{first_element}(\sigma_{\text{min}})$, $t = 0$, $sls_{\text{min}}.MET = (e_i, t) \cup sls_{\text{min}}.MET$

(b) If next($e_i$) = Tick then $t = t + 1$, otherwise $e_i = \text{next}(e_i)$ and $sls_{\text{min}}.MET = (e_i, t) \cup sls_{\text{min}}.MET$

(c) Repeat Step 3.(b) until no more elements left in $\sigma_{\text{min}}$

**Step 4.** Repeat Step 3 for $sls_{\text{max}}.MET$ (replacing $\sigma_{\text{min}}$ with $\sigma_{\text{max}}$).

**Step 5.** $sls*.ET = t. sls*$, where * stays for min or max.

It is to be noted that $sls_{\text{min}}.ET$ represents the minimum duration in time for the scenario $sls$, because of the minimalness property of the set $\{\sigma_{\text{min}} | \exists i \in C \}$, and $sls_{\text{max}}.ET$ represents the maximum duration because of the maximalness property of the set $\{\sigma_{\text{max}} | \exists i \in C \}$.

**Step 6.** $SLS = sls \cup SLS$

**Step 7.** Go to Step 1.

Our methodology is illustrated on the railroad crossing problem introduced in the following section.

6.0 Railroad Crossing Case Study

The Railroad Crossing problem is a bench-mark case study in the real-time reactive systems research community and has been originally described in [3]. In our work we use the version and the time constraints introduced in [1], and illustrate how canonical scenarios are computed for this example. In this version, several trains may cross a gate independently and simultaneously using non-overlapping tracks. A train chooses the gate
it intends to cross, and sends a request to enter the gate. The controlling system monitoring the gate receives the message and instructs the gate to close. The system monitors the gate until no more trains remain to cross the gate, and instructs the gate to open. This cycle of behavior will continue during every period in which trains cross gates. For simplicity, we consider the One-Train/One-Gate case.

- A train indicates its presence to the controlling system with the event "Near".
- The train enters the crossing within an interval of 3 to 5 time units after having sent the event "Near", and sends the event "In" to the system, informing it that it is in the crossing.
- The train informs the system that it is leaving the crossing by sending the event "Exit" within 1 to 3 units of the instant it sends the event "In".
- The system instructs the gate to close by sending an event "Down" within 1 time unit of receiving a "Near" event from the first train entering the crossing.
- The system instructs the gate to open with the event "Up" within 1 time unit of receiving a message "Exit" from the last train to leave the crossing.
- The gate must close within 1 time unit of receiving the event "Down" from the system, and send an event "Closed" to inform the system accordingly.
- The gate must open within an interval of 1 to 2 time units of receiving the event "Up", and send an event "Opened" to the system.

The RDF of the original requirements is given below:

```
ControllerSystem
Train SENDS Near TO ControllerSystem NTR
WHEN Train SENDS Near TO ControllerSystem, Train SENDS In TO ControllerSystem 3, 5
AFTER Train SENDS In TO ControllerSystem, Train SENDS Exit TO ControllerSystem 1, 3
IF Train SENDS Near TO ControllerSystem THEN ControllerSystem SENDS Down TO Gate 0, 1
IF Train SENDS Exit TO ControllerSystem THEN ControllerSystem SENDS Up TO Gate 0, 1
AFTER ControllerSystem SENDS Down TO Gate, Gate SENDS Closed TO ControllerSystem 0, 1
AFTER ControllerSystem SENDS Up TO Gate, Gate SENDS Opened TO ControllerSystem 1, 2
```

In this RDF, for instance, Near is stored as an Input, whereas Down is stored as an Output in the list of messages. The directed acyclical graph is shown in Figure 1.

```
Figure 1. Directed Acyclical Graph for the Railroad Crossing Problem
```
The resultant Input-Output sequence is as follows:

Near → Down → Closed → In → Exit → Up → Opened

The corresponding rules determining the partial order between the environmental events (including the time constraints expressed in terms of Ticks) are specified below:

**Input-Output:**
R1. [Near, Down]
R2. [Near, Tick, Down]
R3. [Exit, Up]
R4. [Exit, Tick, Up]

**Output-Output:**
- Not Applicable

**Output-Input:**
R11. [Down, Closed]
R12. [Down, Tick, Closed]
R13. [Up, Tick, Open]

**Input-Input:**
R5. [In, Tick, Exit]
R6. [In, Tick, Tick, Exit]
R7. [In, Tick, Tick, Tick, Exit]
R8. [Closed, Tick, In]
R9. [Closed, Tick, Tick, In]
R10. [Closed, Tick, Tick, Tick, In]

**Trigger – Last:**
R15. [Near, NTR, Open]
R16. [Near, NTR, Exit]

The algorithm for generating the set $\Sigma$ results in a large set of legal sequences. A sample list of sequences is listed below:
- [Near, Down, Closed, Tick, In, Tick, Exit, Up, Tick, Open]
- [Near, Down, Closed, Tick, In, Tick, Exit, Up, Tick, Tick, Open]
- [Near, Down, Closed, Tick, In, Tick, Exit, Tick, Up, Tick, Tick, Open]
- . . . 

However, the canonical set $C$ generated by the algorithm CSG from the set $\Sigma$ contains only one pair of sequences:

$\sigma_{\min} :$ [Near, Down, Closed, Tick, In, Tick, Exit, Up, Tick, Open]

$\sigma_{\max} :$ [Near, Down, Closed, Tick, Tick, Tick, In, Tick, Tick, Tick, Exit, Tick, Up, Tick, Tick, Open]

The corresponding set SLS of system-level pair of scenarios $sls$ generated for ($\sigma_{\min}$, $\sigma_{\max}$) by the algorithm GSLS is specified below:

- $sls_{\bullet}.SE = \{\text{Near, In, Exit, Up, Down, Open, Closed}\}$
- $sls_{\min}.MET = \{(\text{Near, 0}), (\text{Down,0}), (\text{Closed,0}), (\text{In,1}), (\text{Exit,2}), (\text{Up,2}), (\text{Open,3})\}$
- $sls_{\max}.MET = \{(\text{Near, 0}), (\text{Down,1}), (\text{Closed,2}), (\text{In,5}), (\text{Exit,9}), (\text{Up,10}), (\text{Open,12})\}$
- $sls_{\min}.ET = 4$
- $sls_{\max}.ET = 12$

The formal description of the scenarios is independent of the development environment, therefore allowing for their mapping to software models using different levels of formality (textual, diagrammatic, or a formal specification language).

The scenarios generated in SLS are sufficient to model the critical timing behavior of the system's boundary. The above-mentioned set serves as input to the design specification and the performance
measurement, which is described in the following section.

7.0 Performance Measurement

The environmental events in a scenario are observable, and hence the validation experiment can also 'observe' the satisfaction of time constraints, as specified in the system-level scenario. When an implementation is consistent with the specifications of the system, the execution time in the implementation should conform to the ET specified in the scenario. The response time measure (RTM) is defined as a ratio between the scenarios, the total design execution time of which conforms to the minimum/maximum delay specifications, and to the total number of executed scenarios. This is a simple, absolute scale measure collected from the scenario description and the design/implementation execution trace. The range of the RTM values is $[0..1]$. A value of less than 1 would indicate the existence of an inconsistency between the specified and simulated response times.

8.0 Conclusions and Directions for Future Work

This paper presents a new approach to the generation of formal system-level scenarios from textual requirements written in constrained English, and is aimed at assisting developers in the tedious task of producing correct and complete specifications. The novelty of our approach lies in the systematic and automatic generation of a minimum number of scenarios from a formatted text, completely covering the critical behavior of a real-time reactive system. The methodology allows for objective assessment of the requirements specification phase, and thus can reduce an introduction of human errors early in the software development life cycle. The practical applicability of our methodology is illustrated on a case study. We are currently investigating NLP approaches to processing unrestricted text that would allow for formatting the requirements automatically, and consequently allowing for any textual description of the problem to be dealt with and also reducing the processing time of the requirements.

9.0 References


