Considering the Exceptional: Incorporating Exceptions into Property Specifications

Huong Phan
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
hphan@cs.umass.edu

George S. Avrunin
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
avrunin@cs.umass.edu

Lori A. Clarke
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
clarke@cs.umass.edu

Abstract

Property specifications concisely describe aspects of what a system is supposed to do. It is important that these specifications be correct, describing all the desired behavior and ruling out undesired behavior. Our experience shows that properties are sometimes specified incorrectly because specifiers fail to take into account some exceptional behaviors of the system. In previous work we presented PROPEL*, a tool that guides specifiers through the process of creating understandable yet precise property specifications. Here we describe extensions to* PROPEL *that allow specifiers of a property to indicate what exceptions should be considered and what impact those exceptions should have on the acceptability of system behavior. Although the description is given in terms of the framework provided by* PROPEL*, the issues specifiers must consider would apply to other specification formalisms.*

Keywords: Requirements, Property Specifications, Exceptions, PROPEL.

1 Introduction

Property specifications concisely describe aspects of what a system is supposed to do. These specifications can then be used as the basis for system development and validation. Therefore it is desirable that these property specifications be both formally precise, so that they can serve as the basis for validation, and accessible enough to be easily understood by system developers, who might not be experts in specification formalisms.

PROPEL, for "PROPerty ELucidator", is a tool that guides specifiers through the process of creating understandable yet precise property specifications [4, 10]. PROPEL provides a set of property templates with explicit variations for specifiers to consider, thus ensuring that subtle details are not overlooked. PROPEL allows specifiers to work with three different, but coordinated, views: an interactive question-tree view, a disciplined natural language view, and an extended finite-state automaton view. The first two views do not require specifiers to have expertise in any particular formalism. The last view, on the other hand, provides a precise specification to be used in formal analysis.

From our experiences using PROPEL to specify properties (e.g., [2]), we realized that when properties were specified incorrectly it often was because specifiers failed to take into consideration some exceptional behavior of the system. Exception handling, in fact, is an integral part of most systems, and exception handling mechanisms are directly provided in many programming languages, such as C++ and Java, as well as in some process modeling languages, such as Little-JIL [12]. Very little work, however, has been done to facilitate the specification of exceptions in properties.

In this paper, we describe extensions to PROPEL that overtly take exceptional behavior into account. With these extensions, specifiers can now indicate what exceptions should be considered in a property and decide what impact those exceptions have on the acceptability of the behavior by selecting among variations that PROPEL explicitly makes available. Although the description is shown with respect to the PROPEL framework, the concerns that must be considered would apply to other property specification formalisms.

The next section reviews the property-specification approach used in PROPEL. Section 3 shows example properties that motivate the introduction of exceptions in the specification of properties. Section 4 discusses how exceptions are incorporated in property specifications using PROPEL. Section 5 describes related work, and Section 6 concludes with a discussion about the status of this work, our experience using this approach, and potential areas of future work.

2 Background

PROPEL aims to guide specifiers through the process of creating property specifications that are both accessible and mathematically precise. It is built upon the property patterns developed by Dwyer, Avrunin, and Corbett [6]. Each of the property patterns describes a *behavior* (called an "intent" in the Dwyer et al. work)—a restriction on the occurrences of events or states. For instance, the behavior of the Precedence pattern is an ordered relationship between a pair of events or pair of states where the occurrence of the first (*enabler*) is a necessary pre-condition for an occurrence of the second (*action*). Response (the occurrence of the *action* must be followed by an occurrence of the *response*), Existence (the *action* must occur), and Absence (the *action* must not occur) are some other examples of behavior patterns. A behavior might be intended to hold only while the system is executing in a certain mode, or *scope*. The scopes described in the property pattern work are: Global (the whole execution), Before (execution up to a given event or state), After (the execution after a given event or state), Between (any part of the execution from one given event or state to another given event or state), and After-Until (like the Between scope but the designated part of the execution continues even if the second event or state does not occur). The property pattern work identifies eight behaviors and five scopes that can be combined to create forty different property patterns.

Although the property patterns are very useful, Dwyer et al. admit that most specification formalisms are "a bit tricky to use" [5]. Users are required to have significant expertise with the particular specification formalisms. Dwyer et al. also realize that there are variations of each property pattern, and their website includes notes about some of those variations, although the notes do not attempt to point out all possible modifications that can be made to a property pattern. For example, in the notes about the Precedence pattern, it says that the pattern does not require that each occurrence of *action* have its own occurrence of *enabler*. It is up to specifiers to impose such a requirement if it is desired, thus obtaining a variation of the pattern. There is, however, no guidance on how to do this.

The PROPEL approach makes the specification of properties easier by extending some of these property patterns with *property templates* that explicitly indicate alternative *options* associated with each property pattern. Each of the property templates is composed of a scope template, which contains options related to the selected scope, and a *behavior template*, which contains options related to the selected behavior. PROPEL currently supports variations on all five scope patterns and on four of the behavior patterns identified in the property pattern work: precedence, response, existence, and absence. And although the pattern work includes both event- and state-based forms of the property patterns, PROPEL concentrates only on the event-based forms. Using PROPEL, specifiers instantiate a property by making decisions about, or *resolving*, each of the available options in a selected property template. For example, one option in the Precedence behavior template allows specifiers to determine whether or not each occurrence of *action* must have its own occurrence of *enabler*. To resolve an option, specifiers select one of the option's possible *settings*.

For a property to be *fully instantiated*, not only must all the options be resolved, but specifiers must also define the set of events, or the *alphabet*, that is considered relevant to the meaning of the property. PROPEL distinguishes *primary events*, or "events of primary interest", and *secondary events*, or "other events in the alphabet of the property." Primary events are associated with predefined placeholders, or *parameters*. PROPEL allows up to four different parameters: at most two for the property's behavior (A and B) and at most two for the property's scope ($START$ and END). The parameter names (A, B, $START$, or END) are displayed in the property templates until specifiers associate one or more user-defined events with each displayed parameter. (When two or more events are associated with a parameter, an occurrence of any one of them is treated as an occurrence of the parameter. For example, in a Precedence pattern in which prior to an occurrence of *action* requires an occurrence of *enabler*, the *enabler* can be associated with event E_1 and event E_2, meaning that either E_1 or E_2 can enable the occurrence of *action*.) Specifiers may also define secondary events, not associated with any property template parameters. Secondary events can be used when a property must constrain more that just the occurrences of primary events. For example, it may be important that certain events do not occur in between an occurrence of a primary *enabler* and a primary *action* events. In this case, specifiers can include secondary events in the property's alphabet and define the property so that it explicitly prohibits all occurrences of the secondary events between the *enabler* and *action*.

With the aim of guiding specifiers through the process of creating property specifications that are both accessible and mathematically precise, PROPEL currently provides three different views of the property pattern templates: a Question Tree (QT) view, a Disciplined Natural Language (DNL) view, and an extended Finite-State Automaton (FSA) view. Together they aim to bridge the gap between accessibility and precision in property specification. The QT view provides interactive guidance to specifiers in selecting the desired behavior and scope as well as resolving the related options; the DNL view is accessible to specifiers who are inclined to natural language description; and the FSA view provides the precision that is necessary for finite-state verification. Specifiers can work with any or all of these views. PROPEL automatically maintains the synchronization of the

views, allowing specifiers to switch from one to all or any other views at any point after selecting the desired patterns.

When using PROPEL, specifiers start out by answering initial questions in the QT. The QT view has two separate parts, one for the scope template and one for the behavior template. The initial behavior questions are shown in Figure 1 (the initial scope questions are shown in Figure 6). Answering the initial scope questions and initial behavior questions determines a scope template and a behavior template respectively.

The QT provides an interactive format where specifiers are presented with a question and a choice of possible answers. Only one answer to a question is selected at a time. And based on which answer is selected, a new set of questions and their associated answers may then be presented for further consideration. Because of the QT's hierarchical nature, we refer to those questions as the *child questions*, and the question to which the answer was selected as the *parent question*. The QT hides all the questions that are not relevant to the currently selected answers, thus helping to keep specifiers focused.

Once the desired behavior and scope templates are selected, specifiers can continue working in the QT view or switch to other views to resolve all the options related to those templates.

The DNL view displays *core phrases* for the chosen scope and behavior property templates. A DNL template presents the options related to each core phrase by a short list of *alternative phrases* from which specifiers can choose. In addition, the DNL template view also provides a few *synonyms* to support customization. Figure 7 shows part of the DNL view with a core phase and alternative phrases in the drop-down box for supporting exceptions, as described in Section 4. The DNL view targets specifiers who are unfamiliar with formalisms and/or prefer a natural language description.

The extended FSA is the third view provided by PROPEL. Since the fully instantiated form of this property view is an FSA, this view offers the precision necessary for formal analysis.

Traditionally, an FSA is defined by the tuple $\langle S, s, A, \Sigma, \delta \rangle$ where S is the finite set of states, $s \in S$ is the unique start state, $A \subseteq S$ is the set of accepting states, Σ is the event alphabet, and $\delta : S \times \Sigma \rightarrow S$ is a transition function. An event sequence $e_1 e_2 \ldots e_n \in \Sigma^*$ is accepted by the FSA if a sequence of states $s_0, s_1, \ldots, s_n$ exists in S such that: (1) $s_0 = s$; (2) $s_n \in A$; and (3) $\delta(s_{i-1}, e_i) = s_i$ for $i = 1, \ldots, n$.

In addition, PROPEL requires that the FSA be total and deterministic; that is every event in the alphabet must be associated with one and only one transition emanating from each state. Often a single non-accepting trap state is introduced so that transitions to this trap state can be added to satisfy this requirement.

When shown graphically, FSA states are shown as circles, the start state is denoted by an arrowhead on the circle, an accepting state is indicated by an inner concentric circle, and a transition is denoted by an arrow between two states indicating the direction of flow in the automaton. Each transition is labeled by one or more events from the alphabet.

In PROPEL, the FSA notation is extended so that an FSA template can be displayed with options to be resolved. An option is represented in one of the following forms: *an optional transition*, denoted by a dashed line instead of a solid line; *an optionally-accepting state*, denoted by a dashed inner concentric circle; and *a multi-label*, denoted by a list of alternative sets of labels, each set separated by the word "or". Specifiers resolve the options by making decision about: whether an optional transition should exist; whether an optionally-accepting state should be accepting; and which of the label choices should be selected in a multi-label.

Other symbols are added to the extended FSA template notation in order to improve the clarity of this view: "$\neg$" (the set complement operator) and "." (the wildcard character, representing all of Σ).

For clarity, when there are no options associated with the non-accepting trap state or with any transitions to that state, we suppress it and all the transitions to it.

Once specifiers select the desired behavior pattern and chosen to work with the FSA view, the corresponding FSA template will be shown with all the options to be resolved. The states in the extended FSA view of the behavior part of the property are called *behavior states*. The scope part of the property is incorporated after specifiers choose the desired scope pattern with the QT. The scope is applied to the behavior by adding additional states, called *scope states*, and by adding additional transitions between the scope states and the behavior states. These transitions are labeled by the scope delimiter events. Partially and fully instantiated FSA views of an example property are given in Figure 2 and discussed later in this section.

PROPEL identifies two special states in the FSA view: a *behavior start state* and a *behavior violation state*. A behavior start state is the start state of the FSA representing the behavior part of the property, before applying the scope. A behavior violation state, which is often shortened to *violation state*, is a non-accepting trap state in the behavior part of the property. That state has only one emanating transition, a self-loop transition with labels for the entire alphabet, i.e. once the FSA is driven to this state, it stays there. Note that although the violation state is non-accepting, not all non-accepting states are violation states. Moreover, not all properties have violation states.

Consider the following property of a blood transfusion process, written in natural language: *"After the physician*

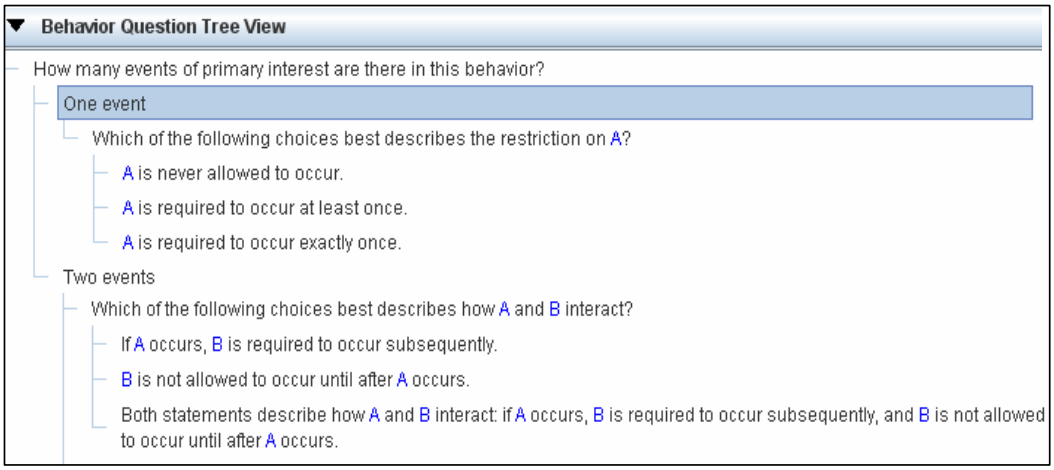

Figure 1. Initial behavior questions and answers

has ordered a blood transfusion for a patient, the presence of a signed-consent form must be confirmed before infusing a single unit of blood product to the patient." This property's alphabet contains three events. To avoid clutter in displaying the property, we assign each event a short name: **receive-order** for "the physician orders a blood transfusion for the patient"; **consent-confirmed** for "the presence of a signed consent form is confirmed"; and **infuse-blood** for "the infusion of a single unit of blood product is started."

At first glance, it is easy to see this property's scope maps to the After pattern (the event **receive-order** marks the beginning of the execution of interest). The property's behavior maps fairly well to the Precedence pattern in which an *action* is not allowed to occur until after an *enabler* event has occurred. The enabler event would be associated with the event **consent-confirmed** and the action event with **infuse-blood**.

Figure 2 shows the FSA view of this property: one partially and one fully instantiated. In the partially instantiated FSA, the following options are not yet resolved: (i) whether or not an *action* event (**infuse-blood**) is allowed to occur again after the first occurrence of an *action*; (ii) if an *action* event is allowed to occur again, whether or not it requires its own occurrence of an *enabler* event (**consent-confirmed**). The options are presented by optional transitions and a multi-label of the self-loop transition at state 4 (here we label the states with numbers for easy reference). The fully instantiated FSA shows that both the options are set to *true*. State 1 is the scope state, the rest are behavior states. State 2 is the behavior start state. State 5 is the behavior violation state. An event sequence will drive the FSA to this behavior violation state if each **infuse-blood** event is not preceded by a **consent-confirmed** event, i.e. blood is infused without confirming the presence of the signed consent form.

3 The need to specify exceptional events

In our experience specifying properties and verifying them using finite-state verification, we found that one of the main reasons properties were incorrectly specified was that specifiers failed to take into account how some *exceptional events* might impact what is considered correct or incorrect system behavior. In this section we illustrate this with some examples of properties that were initially specified incorrectly due to lack of consideration of exceptional events. These properties are all concerned with the administration of a single blood transfusion to a patient who arrives in the Emergency Department of a hospital, is registered into the hospital's computer system, and is determined by a physician to need a blood transfusion.

First we look at an example of a property that deals with a temporary interruption in a transfusion:

> ***Property 1a:*** *After the infusion has started, if it has been interrupted due to a suspected transfusion reaction, it must be resumed.*

This property can be specified using the After scope pattern and the Response behavior pattern, for which a *stimulus* event must be followed by a *response* event. In this case, **interrupt**, short for "the infusion is interrupted due to a suspected transfusion reaction", is the stimulus event; **resume**, "resume the infusion", is the response event; and **infusion-start** is the scope guarding event, marking the start of the execution of interest. After the occurrence of **infusion-start**, if **interrupt** occurs but **resume** never occurs, then the situation is not acceptable according to this property specification. In fact, during the blood transfusion process, if the physician cancels an order for an infusion of a unit of blood product, then of course that infusion does not need to

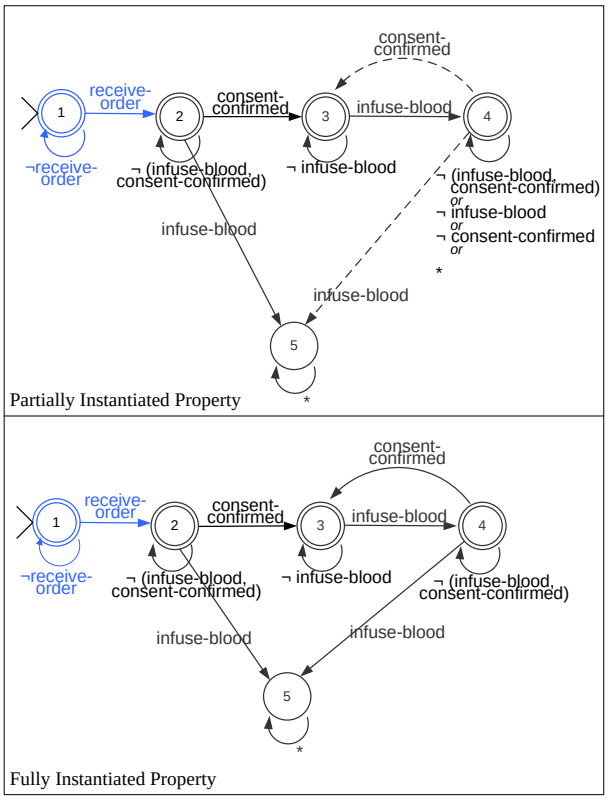

Figure 2. Example property expressed using the After scope and the Precedence behavior

be resumed after having been interrupted. The event **cancel**, for "the physician cancels the order for the infusion" is an exceptional event that must be taken into account. After consulting with a domain expert, we revised the informal statement of property 1a to the following:

> ***Property 1b:*** *After the infusion has started, if it has been interrupted due to suspected transfusion reaction, it must be resumed unless the physician cancels the order for the infusion.*

Careful consideration of this exceptional event leads the specifier to decide that if this exceptional event occurs anywhere in a trace through the process, then that trace should be considered to be acceptable, that is, considered to be consistent with the stated property. (Other properties would specify, for instance, that if the **cancel** event occurs before the infusion is started, that infusion should never be started, etc.)

Let us now consider another property associated with the patient registration process. This process defines the steps that need to be done to register a patient into the hospital's computer system. The process starts when the patient enters the hospital and ends when a patient ID band is placed on the patient's arm. In this process, the clerk must ask for the patient's name, date of birth, etc. One of the properties is:

> ***Property 2a:*** *Before printing the patient ID band, the patient's name must be obtained.*

The property in fact can be expressed in two ways, using the Existence behavior and the Before scope patterns, or using the Precedence behavior and the Global scope patterns. For the purpose of this discussion, we will express this property using the Existence behavior and the Before scope patterns. The primary event of interest is **pt-name-obtained**, for "patient name obtained". A trace through the process is considered to satisfy the property if the event **pt-name-obtained** occurs within the scope before the scope guarding event **print-ID-band**. Otherwise, if the event **pt-name-obtained** does not occur within the scope, that trace is considered to violate the stated property 2a.

In an Emergency Room setting, however, it is sometimes the case that patients arrive unconscious, carrying no identifying information, and without someone to identify them. Thus it is impossible to obtain the patient's name before starting treatment. In that case, according to the hospital's Patient Identification Protocol, the patient is still registered in the system, but given a name such as John/Jane Doe, so that medical care for the patient can proceed. In such a trace through the process, the event **pt-name-obtained** obviously never occurs. This trace should still be considered acceptable though it does not satisfy property specification 2a. Revision of the property specification is needed; the exceptional event **unable-to-obtain**, for "unable to obtain patient's identity", must be taken into account. A revised informal statement of property 2a is:

> ***Property 2b:*** *Before printing the patient ID band, the patient's name must be obtained unless the patient is unconsciousness and other sources to obtain the patient's identity are not available.*

(Of course, another property must be specified to describe the appropriate actions in the case where it is impossible to obtain a patient's name, but the details of that property are not relevant here.)

The impact of the exceptional event in this case is somewhat different from the impact of property 1b's exceptional event. In property 1b, a trace through the process is considered to satisfy the property if the exceptional event **cancel** occurs at any point in that trace. In property 2b, a trace through the process is considered to satisfy the property if the exceptional event **unable-to-obtain** occurs within the specified scope, i.e. before the event **print-ID-band**. The following situation is certainly not acceptable: the clerk does not obtain the patient's name, enters an arbitrary name

into the system, prints the ID band, and only checks afterward that it is impossible to determine the patient's identity. In this case, a trace with the exceptional event **unable-to-obtain-patient-identity**, occurring after **print-ID-band**, i.e. outside of the scope, cannot be considered to satisfy the property.

Now we revisit the property mentioned in the previous section.

> ***Property 3a:*** *After the physician has ordered a blood transfusion for a patient, the presence of a signed-consent form must be confirmed before infusing a single unit of blood product to the patient.*

This property can be represented with the Precedence behavior and the After scope pattern as shown before. The scope guarding event is **receive-order**. The two events of primary interest are **consent-confirmed** (the enabler) and **infuse-blood** (the action). The action cannot occur until after the enabler has occurred. In an emergency situation, however, when the physician decides that the patient must have the infusion but the consent form cannot be signed (due to the fact that the patient is unconscious), this rule can be bypassed. Following is a revised informal statement of property 3a:

> ***Property 3b:*** *After the physician has ordered a blood transfusion for a patient, the presence of a signed-consent form must be confirmed before infusing a single unit of blood product to a patient, unless the patient is unconscious and the physician decides that the patient must have the infusion, in which case the consent form can be waived.*

That is, if the exceptional event **unconscious** (short for "the patient is unconscious and needs blood infusion") occurs, the event **infuse-blood** can occur without a prior occurrence of the event **consent-confirmed**. More careful consideration reveals that the exceptional event only has such impact when it occurs within the scope but before a behavior violation has been determined. If an infusion is carried out without a signed consent form and the patient is found to be unconscious, but only after the transfusion has started, the trace must be considered to be unacceptable.

In the following section we describe an approach for incorporating exceptional events into property templates in PROPEL, that helps specifiers specify exceptional events and indicate what impact those exceptional events are to have on the system behavior.

4. Incorporating Exceptional Events

Having seen several examples of properties that require consideration of exceptional events, we decided that, to be consistent with the PROPEL philosophy of guiding users through the specification of properties by explicitly indicating the options associated with common patterns, PROPEL should also include the options associated with exceptional behavior. This requires specifiers to indicate the exceptional events that are to be considered and to select the settings for the options associated with that set of events.

Similarly to how specifiers can assign events to the property's behavior parameters, A and B, and to the property's scope parameters, $START$ and END, events can now be assigned to an exceptional event parameter, called X. And, as with behavior and scope parameters, if two or more events are associated with the parameter, an occurrence of any one of them is regarded as an occurrence of the parameter.

As the examples in the previous section show, occurrences of an exceptional event at different locations in an event sequence may have different effects on whether the event sequence is accepted. Thus, after declaring the exceptional events that should be taken into account, specifiers have to resolve the following two independent options:

1. *within-scope*: If this option is set to true, then only exceptional events occurring within the scope are taken into consideration. That is, if exceptional events occur only outside of the scope, the event sequence's acceptability is the same as when the exceptional events do not occur at all. In the case of a property with Global scope, this option *within-scope* is ignored.

2. *before-violation*: If this option is set to true, then only exceptional events occurring before any behavior violation has been found are taken into consideration.

Below, we describe how these options are represented in the FSA, QT, and DNL views.

4.1. FSA View

To support the treatment of exceptional events that result in an event sequence being considered accepted, PROPEL adds a new accepting state, called the *exception state*, to the existing FSA templates and adds transitions on exceptional events from existing states to this exception state. The idea is that once the FSA is put in the exception state, it stays in that state until the end of the event sequence or until a guarding event of the designated scope is encountered.

The treatment of exceptional events must take into account the property's designated scope. For scopes with a

start guarding event (i.e., After and Between) one previously existing option must be considered, *subsequent-start-reset*. This option determines whether or not a subsequent occurrence of a start guarding event resets the scope. The exception state has up to three *out-going* transitions, depending on the scope and the setting of *subsequent-start-reset*:

- If the scope has a start guarding command, there is an optional transition on the start guarding events to the behavior start state. The existence of this transition is determined by the option *subsequent-start-reset*. If this option is set to true, then a subsequent occurrence of the start guarding event resets the scope and this transition exists in the corresponding FSA representation; otherwise this transition does not exist.

- If the scope has an end guarding event, there is a transition on the end guarding event to the accepting trap state.

- There is a self-loop transition with a multi-label. Since the existence of the above transitions depends on the option *subsequent-start-reset* and on whether the scope has an end guarding event, the multi-label on this self-loop transition must reflect these dependencies accordingly. The possible multi-labels are shown in Table 1.

We represent the two exceptional-event-related options (*within-scope* and *before-violation*) by the optional transitions *to* the exception state. The setting of the option *within-scope* is determined by the existence of the transitions from the scope states to the exception state. These transitions exist if and only if the option within-scope is set to false. There is one special case, however, where the scope has an end delimiter. More details are given when we explain property 2b below.

The setting of the option *before-violation* is determined by the existence of the transition from the behavior violation state, the non-accepting trap state in the behavior FSA, to the exception state. This transition exists if and only if the option *before-violation* is set to false.

Let us look at the three properties 1b, 2b and 3b mentioned in the previous section. The Figures 3, 4, and 5 first show the fully-instantiated property, respectively, without exceptional events, then show the FSA template taking exceptional events into account, and finally the fully-instantiated property. For clarity, we assume that all the scope and behavior option settings in properties 1b, 2b, and 3b, including the *subsequent-state-reset* option, are already resolved exactly the same as in properties 1a, 2a and 3a respectively.

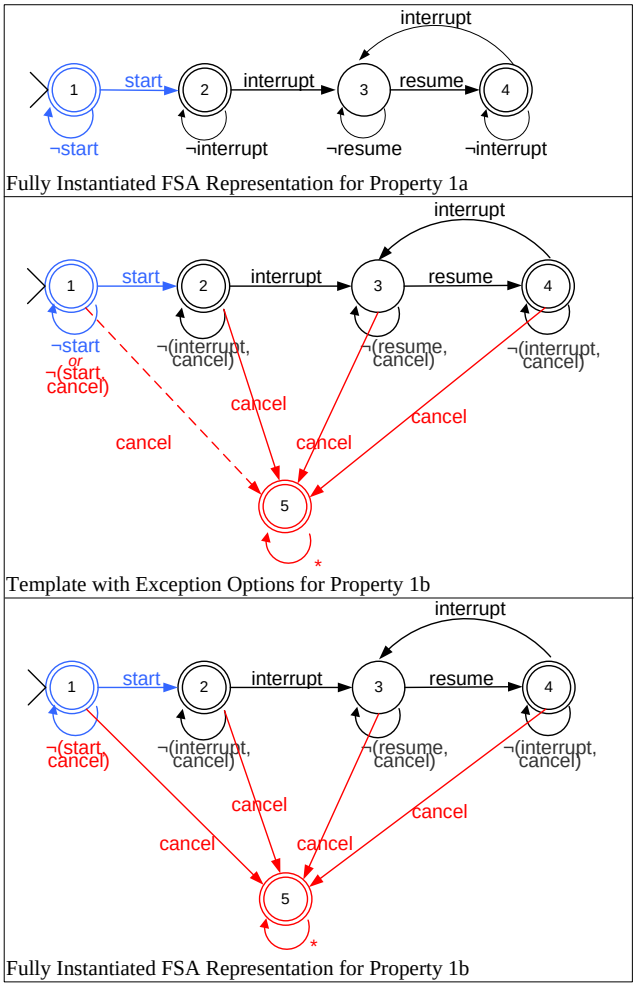

Figure 3. Property 1a and 1b

As shown in Figure 3, the exceptional event **cancel** (the physician cancels the order for the infusion) is incorporated in property 1b by:

- adding the exceptional state (state 5),

- adding additional transitions on **cancel** from all the behavior states to the exceptional state,

- adding an optional transition from the scope state (state 1) to the exceptional state. This transition exists if and only if the option *within-scope* is set to false.

- changing the existing multi-labels to reflect these additions where possible.

Note, we omit the optional transition from the exceptional state to the behavior start state (state 2). That optional transition involves the scope option *subsequent-start-reset*, which in this case was already resolved to *false*. Therefore the fully-instantiated FSA does not show that transition.

Table 1. Multi-label on the Self-loop Transition of the Exception State

Scope	Multi-label	Description
Global	.	A subsequent occurrence of an event in the alphabet while the FSA is in the exception state will not cause a change in state.
After (scope has start guarding event)	$\neg start$ or .	If $\neg start$ is used, a subsequence occurrence of start resets the scope. If . is used, a subsequence occurrence of start does not reset the scope (e.g., property 1b).
Before (scope has end guarding event)	$\neg end$	An occurrence of end while the FSA is in the exception state puts the FSA in the accepting trap state, otherwise there is no change in state(e.g., property 2b).
Between (scope has start and end guarding events)	$\neg(start, end)$ or $\neg end$	If $\neg(start, end)$ is used, a subsequent occurrence of start resets the scope. If $\neg end$ is used, a subsequent occurrence of start does not reset the scope.

As discussed in the previous section, for property 1b, an event sequence is considered acceptable if the exceptional event **cancel** occurs anywhere in the event sequence. Thus the option *within-scope* should be set to false, which leads to the existence of the transition from state 1 to state 5. Resolving this option results in the fully-instantiated property shown in Figure 3.

Exceptional events are incorporated in properties 2b and 3b in the same way. Figure 4 illustrates the case where the property's scope has an end delimiter, **print-ID-band**, and therefore there is an accepting trap state (state 3). When adding the exception state (state 5), a transition on the end delimiter from the exception state to the accepting trap state is added. Property 2b also illustrates the case where the option *within-scope* is set to true; thus there is no transition from the scope state (state 4) to the exception state (state 5). As mentioned above, this is a special case where the scope has an end delimiter, thus state 4 is a scope state that is a *non-accepting trap state* (different from the behavior violation state, which does not appear in this case). An event sequence puts the FSA into state 4 if the scope-ending event is encountered after a behavior violation was found. At this point, an occurrence of an exceptional event is obviously outside of the scope and after the violation. If the event sequence, taking into account the exceptional event, should still be considered accepted, then the existence of the transition from state 4 to state 5 should mean setting both options within-scope and before-violation to false. Therefore, in the case where the scope has an end delimiter, the optional transition from the non-accepting trap state to the exception state exists if and only if both exception-related options are set to true.

Unlike the previous two cases, Property 3b in Figure 5 has a behavior violation state (state 5). The option *before-violation* is represented by the optional transition from the violation state to the exception state (state 6). This transition exists if and only if the option *before-violation* is set to false. It was previously decided that for this property an event sequence is considered acceptable if the exceptional event occurs within the defined scope and before a behavior violation is found. The options *within-scope* and *before-violation* are, therefore, both set to true. The optional transitions from the violation state (5) and the scope state (1) are therefore removed.

The changes to the QT and DNL views to support exceptional events are much more straightforward and show the benefit of having carefully-designed natural-language views. By making a few explicit decisions in these views, all the options shown in the FSA view are decided.

4.2. Question Tree View

In the QT, a new question (Q1) is added to let specifiers indicate whether or not any exceptional events should be considered. This question is placed in the scope QT, as a child question after specifiers select the answer for the initial scope question. By answering "Yes" to Q1, specifiers indicate that some exceptional events must be taken into account. The second new question (Q2) lets specifiers resolve the exception-related options. Figure 6[1] shows the case where the specifier chooses a non-Global scope: after the specifier indicates the need to consider exceptional events, a sentence opening "The event sequence is considered acceptable if X occurs" is presented and the specifier must select among the following four choices to complete the sentence to describe the impact of the exceptional events:
(1) anywhere in the event sequence.
(2) anywhere in the event sequence before any behavior violation has been found.
(3) within the designated scope.
(4) within the designated scope and before any behavior violation has been found.

[1] In the figures, X is the parameter's default name that will be replaced by the user-defined exceptional event names once the specifier associates exceptional events with this parameter.

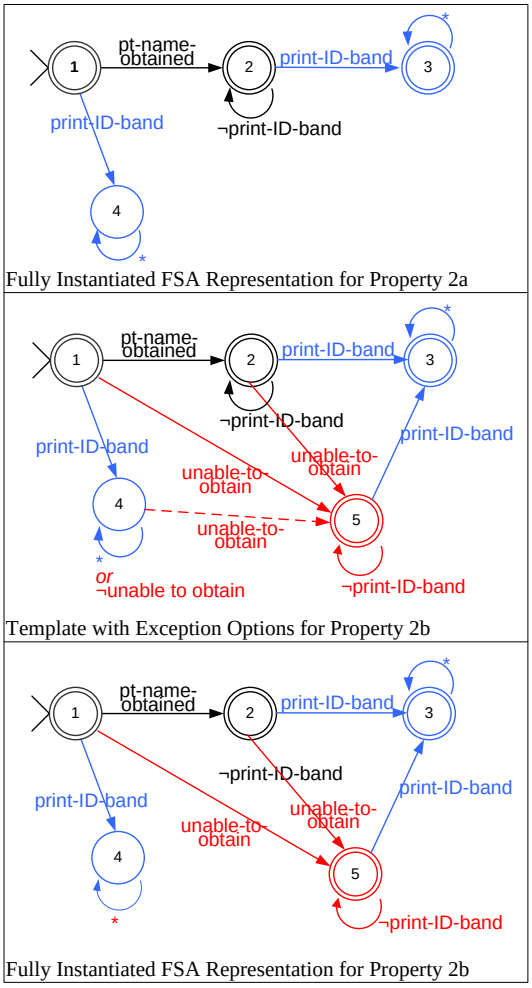

Figure 4. Property 2a and 2b

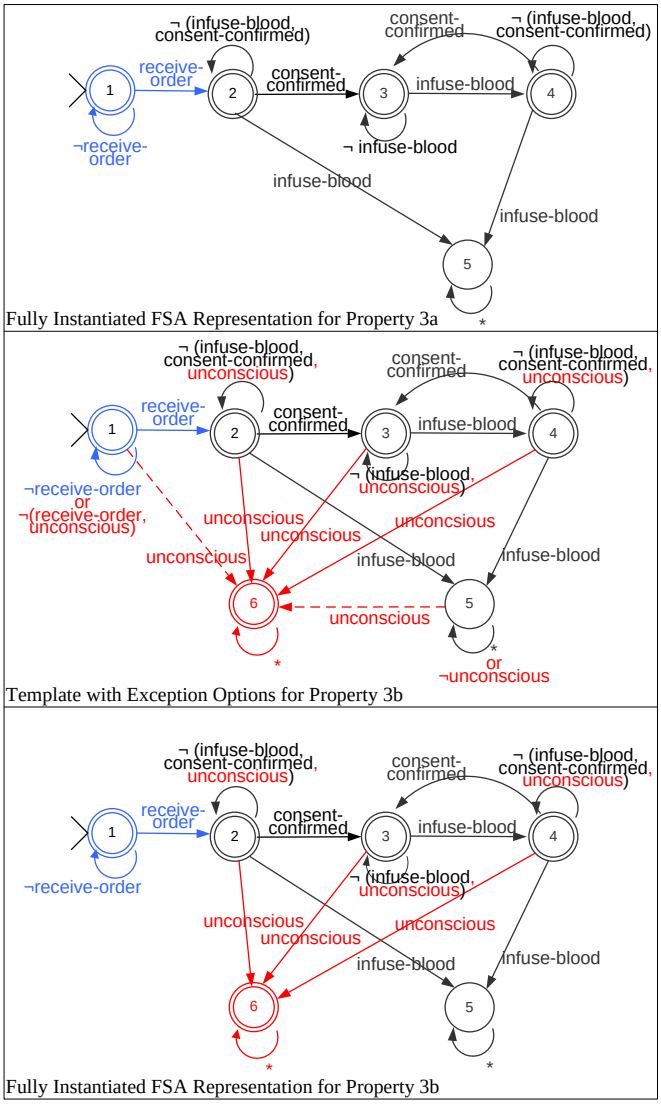

Figure 5. Property 3a and 3b

In the case of the Global scope, since option *within-scope* is ignored, only the first 2 choices would be displayed to the specifier.

4.3. Disciplined Natural Language View

The DNL view is also updated to allow specifiers to resolve the exceptional-event-related options. Similar to the way the alternative answers are presented in the QT view, a sentence composed of a core phrase and four alternative phrases are added to the DNL view in the case of a non-Global scope (Figure 7), while in the case of the Global scope, the sentence is composed of the same core phrase and only the first two alternative phrases. The corresponding option settings are the same as those in the QT view.

5 Related work

There is a considerable body of work attempting to bring accessibility and understandability to formal property specifications, including a number of papers that extend the property patterns of Dwyer et al. Due to space limitations, we do not summarize that work here (but refer the reader to [4] and [3] for such discussions) and instead focus on the work related to exceptions.

The idea of exceptions in programming languages has a long history, going back to the 1970s (see [9], for example, for a discussion), but the only work we are aware of that discusses the impact of exceptions on property specifications like those produced with PROPEL is that of Flake and Mueller [7]. They proposed a temporal extension to

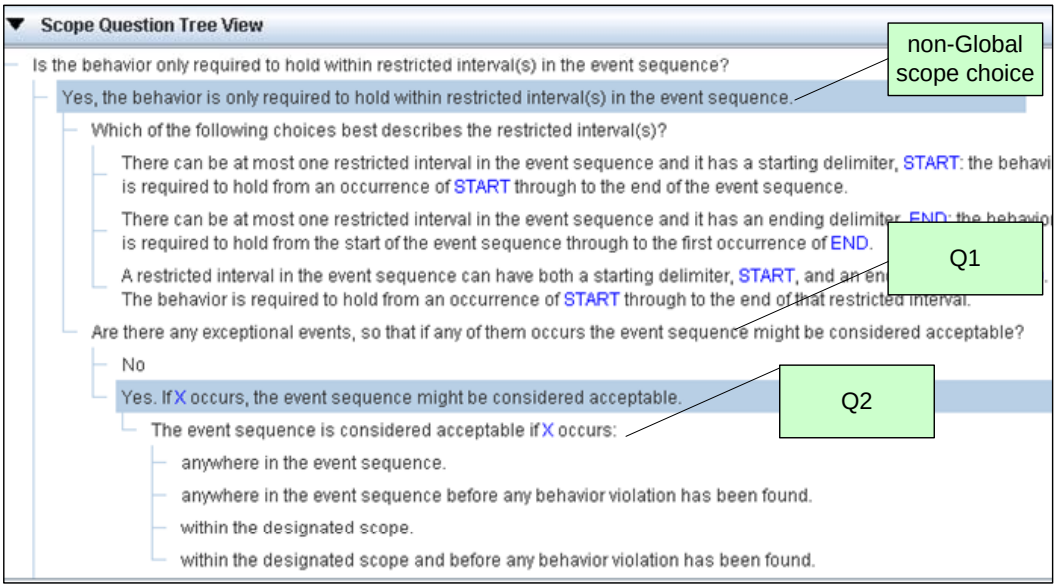

Figure 6. Exceptional-event-related Questions and Their Answers for a non-Global Scope Property

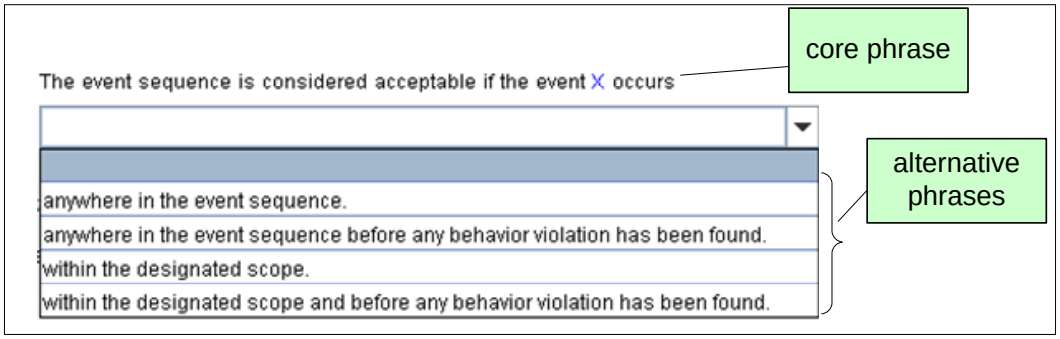

Figure 7. Exceptional-event-related DNL Phrases for a Non-Global Scope Property

the Object Constraint Language (OCL) to allow the expression of a modified version of the property patterns for UML Statecharts. They also pointed out that their approach could be combined with the approach of Soundarajan and Fridella for specifying exceptional situations in OCL [11] by specifying appropriate post-conditions for methods. They do not, however, seem to have implemented this. The general idea of specifying erroneous or exceptional behavior by post-conditions can be traced back to OBJ [8], and a similar approach is used, for example, in JML [1], but none of these approaches provides the sort of guidance for specifiers in describing the impact of exceptional behavior on acceptability that PROPEL does.

6 Conclusion

The decision to incorporate support for exceptional events arose based on our experiences eliciting and representing properties for medical processes [2]. As remarked above, we noticed that the most common reason that properties were specified incorrectly was because they failed to take exceptional conditions into account. We discovered these problems, not when eliciting or reviewing the properties with domain experts, but while we were trying to verify process definitions of medical processes. All of the properties that failed because of this problem can now be represented appropriately using the current version of PROPEL that provides support for exceptional events as described in this paper. In all these cases, the exceptional events are actually associated with raised exceptions in the process definitions, although that does not always have to be the case–conditions that the specifier views as exceptional need not

be explicitly represented that way in the system whose requirements are being specified.

This version of PROPEL also provides support for the disjunction of events, as described in this paper. It is sometimes the case that disjunction can be used to support exceptional events. For example, Property 1b could be restated as: After the infusion has started, if it has been interrupted due to suspected transfusion reaction, it must be resumed or canceled. Disjunction does not provide support for all the four cases that need to be supported, however, and can lead to awkward representations.

There are several areas of future research. One interesting issue is how exceptional events should be supported during finite-state verification. For example, what if a property is found to be consistent with a program but all the traces through that program are acceptable because the property is always driven to the accepting exception state. Certainly verifiers would want to be informed of this situation since presumably the non-exceptional behavior was expected to occur.

There are some extensions to the current version of PROPEL that should be investigated. Some of the original property patterns are not supported. We have also noticed that an alternation pattern between two events, where either one can occur first, occasionally is needed and is not currently supported by PROPEL or the original property patterns. The current version of PROPEL also treats all the events associated with a parameter or with secondary events the same; they are represented as disjuncts and, thus, in the FSA are shown as multi-labels on the appropriate transitions. It might be reasonable to allow specifiers to select among these labels. In considering possible extensions, however, our goal is not to support all possible property specifications but to support the more common types of properties that occur in practice. Based on our case studies to date [4, 3], this has indeed been the case, with PROPEL naturally supporting over 90% of the properties that have arisen.

Finally, more experimental evaluation needs to be undertaken. We basically retrofitted the problematic properties from our studies using the PROPEL extensions for supporting exceptional events. We do not know if these exceptional events would have been taken into account in the original property formulation if the exceptional options had been presented to the domain experts. Nor do we know if significantly more properties will take exceptional events into consideration if these options are consistently presented to the specifiers and domain experts. It would also be interesting to see if these extensions influence specifiers developing properties for programs written in Java or other programming languages that provide explicit support for exceptional conditions.

References

[1] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[2] B. Chen, G. S. Avrunin, E. A. Henneman, L. A. Clarke, L. J. Osterweil, and P. L. Henneman. Analyzing medical processes. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 623–632, New York, NY, USA, May 2008. ACM.

[3] R. L. Cobleigh. PROPEL*: An Approach Supporting User Guidance in Developing Precise and Understandable Property Specifications*. PhD thesis, University of Massachusetts Amherst, 2008.

[4] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke. User guidance for creating precise and accessible property specifications. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering*, pages 208–218. ACM, 2006.

[5] M. Dwyer, G. Avrunin, and J. Corbett. Property specification patterns website. http://patterns.projects.cis.ksu.edu/.

[6] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, 1999.

[7] S. Flake and W. Mueller. Expressing property specification patterns with OCL. In *International Conference on Software Engineering Research and Practice*, pages 595–601. CSREA Press, 2003.

[8] J. A. Goguen and J. J. Tardo. An introduction to OBJ. In M. V. Zelkowitz, editor, *Proceedings of SRS, Specifications of Reliable Software*, pages 170–189. IEEE, 1979.

[9] B. G. Ryder, M. L. Soffa, and M. Burnett. The impact of software engineering research on modern progamming languages. *ACM Trans. Softw. Eng. Methodol.*, 14(4):431–477, 2005.

[10] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Propel: an approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 11–21. ACM, 2002.

[11] N. Soundarajan and S. Fridella. Modeling exceptional behavior. In *In UML99 The Unified Modeling Language. Beyond the Standard. Fort Collins, CO, USA, volume 1723 of LNCS*, pages 691–705. Springer, 1999.

[12] A. Wise, A. Cass, B. Lerner, E. McCall, L. Osterweil, and J. Sutton, S.M. Using Little-JIL to coordinate agents in software engineering. *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 155–163, 2000.