Domain-Specific Language for Crisis Management Systems

Nadin Bou Khzam

Department of Electrical and Computer Engineering
McGill University, Montreal

August 2019

A thesis submitted to McGill University in
partial fulfillment of the requirements of the degree of Master of Engineering

Abstract

Across the world, various crisis situations occur causing chaos and confusion as to how to deal with them. Hence, finding ways to transmit the necessary information regarding how to handle such incidents to various parties involved in such events is fundamental. As such, the field of crisis management focuses on determining the actions to undertake to quickly respond to the occurrence of a disaster. Ultimately, crisis management systems strive to guide individuals to better prepare themselves for any future encounter of crisis situations. In this thesis, we investigate emergency circumstances, mainly natural disasters, to detail the requirements for workflow notations managing them. Taking into account these requirements, we propose a domain-specific software language (DSL) for crisis management based on the textual implementation of the Use Case Maps (UCM) metamodel via the Textual User Requirements Notation (TURN). While workflow notations do exist that can model the basic procedures to undergo when a crisis incident occurs, none focuses specifically on crisis management systems or considers the 3D environment in which emergency situations unfold. The proposed DSL provides built-in support for modeling location-based, social media-inspired interactions in the 3D environment of a disaster, allowing crisis experts to author emergency procedures. The models created from the DSL are subsequently traversed using an algorithm we design to step through the workflow to be followed by victims and first responders during crisis situations. As our proof-of-concept implementation, an Android mobile application allows a user to go through crisis management workflows step-by-step in forward or backward direction to validate our findings and work, particularly with respect to the UCM traversal algorithm. The long-term goal is to reduce the damages or losses, in terms of property and life, caused by a crisis, benefiting us all as well as the environment we live in.

Abrégé

À travers le monde, diverses situations de crise font l'objet de chao et de confusion quant à la façon de les gérer. Par conséquent, il s'avère fondamental de trouver des moyens de transmettre les informations indispensables à la façon d'agir dans de telles circonstances aux différentes parties impliquées. Ainsi, le domaine de la gestion de crise se concentre sur la détermination des actions à entreprendre afin de réagir rapidement à une catastrophe. Ultimement, les systèmes de gestion de crise s'efforcent de guider les individus pour mieux les préparer à toute future situation de crise. Dans ce mémoire, nous étudions les situations d'urgence, principalement les catastrophes naturelles, afin de détailler les exigences relatives aux notations de flux de travail qui les gèrent. Tenant compte de ces exigences, nous proposons un langage logiciel dédié (DSL) pour la gestion de crise basé sur l'implémentation textuelle du métamodèle de plan de cas d'utilisation (UCM) via la syntaxe textuelle de la Notation des prescriptions utilisateur (URN), appelée TURN. Bien qu'il existe des notations de flux de travail pouvant modéliser les procédures de base à suivre en cas de crise, aucune ne se concentre spécifiquement sur les systèmes de gestion de crise ni ne considère l'environnement 3D dans lequel se déroulent les situations d'urgence. Le DSL proposé permet la modélisation des interactions inspirées des médias sociaux basées sur la localisation dans l'environnement 3D d'une catastrophe, permettant ainsi aux experts en situation de crise de créer des procédures d'urgence. Par la suite, les modèles créés à partir du DSL sont parcourus à l'aide d'un algorithme que nous avons conçu dans le but de naviguer le procédé à suivre par les victimes et les secouristes en cas de crise. Pour notre preuve de concept, une application mobile Android permet à un utilisateur de traverser les flux de travail de gestion de crise de chaque pas vers celui ou ceux qui le suivent ou le précèdent, afin de valider nos découvertes et notre travail, en particulier en ce qui concerne l'algorithme de navigation de modèle UCM. L'objectif à long terme est de réduire les dommages ou les pertes, en termes de propriété et de vie, causés par une catastrophe, au bénéfice de tous, ainsi que de l'environnement dans lequel nous vivons.

Acknowledgements

I would like to express my deep appreciation of everyone who played a part in the successful accomplishment of my Master's degree, without whom this achievement would not have been possible.

To begin, a special thanks goes to my supervisor, Professor Gunter Mussbacher, for his constant guidance, support, and understanding during my Master's research and throughout the completion of my thesis. His welcoming attitude, his availability and patience for answering questions, and his valuable insight were crucial to the success of my thesis. I feel very fortunate to have been able to work under his supervision.

Second, I am sincerely grateful to my family who stood by me, and provided me with continuous moral support and motivation throughout my Master's degree. I am forever appreciative for their positive and encouraging outlook which allowed me to thrive in this endeavor.

Moreover, I would like to thank the friends I made at the Software Engineering lab for their support and empathy as we worked on our theses alongside each other.

Furthermore, I thank my friends, who were always there to listen and cheer me on during my two years in graduate school.

Lastly, a warm thanks goes to Ruchika Kumar, whose thesis' Textual User Requirements Notation I extended, and Thomas Degueule, who helped me in finding solutions to the Eclipse Modeling Framework and Xtext issues I encountered during the early stages of my thesis implementation.

Thank you.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Due to their unpredictable nature, crisis situations, which may occur anywhere and anytime with often severe after-effects, cause chaos as to how to handle them. As a result, researchers strive to find alternatives of conveying the knowledge critical to dealing with such situations such that the latter may be handled adequately and rapidly by the collaboration of all parties involved. Crisis management is the service sector concerned with defining the tasks that must be accomplished to quickly respond to the occurrence of a disaster. Crisis management systems were thus created for the purpose of guiding societies during crises and better preparing them for future encountered emergency incidents, in hopes of mitigating the extent of the damages entailed by such events [3].

1.1 Motivation

Crisis respondents refer to predefined emergency procedures, encompassing ordered measures to take during specific activities in a crisis [4]. These procedures are normally found in printed documents, thereby making them inefficient and demonstrating the need for a software alternative to model, execute, and manage emergency plans [4]. Use cases and business procedure documents can be hard to comprehend and contain errors at times, resulting in the need for formal graphical workflow notations, such as the Business Process Model and Notation (BPMN) [5], Unified Modeling Language (UML) [2], and User Requirements Notation (URN) [6], to represent processes [7].

With the increasing importance of the service sector in today's society, more attention has been given to the flow of information within system workflows to provide more efficient means of offering services [8]. Workflow systems allow successful solutions to

problem domains [8]. Crisis management systems may be described by workflow notations as the processes they follow satisfy the characteristics of "workflowable" processes which include that they must be clearly outlined, represent repeated events, and involve many parties [8]. Extending an existing workflow notation with domain-specific concepts allows the representation of a particular domain [9]. Since emergency plans and business processes share many characteristics, the former can also be modeled using workflow notations, where communication of information and collaboration must be considered [4], as they represent critical aspects of crisis relief tasks [3].

Since building software systems for specific domains is often difficult, domain-specific languages (DSLs) have been used over the years to fulfill the needs of given application domains [10]. Standard languages, such as BPMN, UML, and URN, are very common today but sometimes they must be extended by adapting their underlying metamodel to meet the demands of a specific domain [11]. The core of a workflow notation remains valid while the concepts specific to a domain are introduced into the metamodel of the notation [11]. Modeling is successful when the extended metamodel of a workflow notation is able to express the needs of the business or system at hand [12].

1.2 Contributions

Over the years, researchers realized the "need to tailor and extend URN for specific application domains" [13]. In this thesis, we propose a crisis management DSL by extending a general-purpose workflow language, the URN's Use Case Maps (UCM) [6], with key crisis management concepts [14]. Specifically, we extend the textual implementation of UCM, Textual Use Case Maps (TUCM) from the Textual User Requirements Notation (TURN) [1], rather than its URN graphical notation. This decision to extend URN's UCM lies in the experience that we have with this notation. However, the key crisis management concepts introduced into URN could also be applied to other workflow notations. Thus, the presented results are not limited to URN.

Using our crisis management DSL, we create TUCMs representing crisis scenario

workflows. We then run our designed traversal algorithm on an instance model of the TURN metamodel to allow the step-by-step processing of the crisis management progress in forward and backward direction [15].

Although existing notations may model the basic procedures to follow during a crisis situation, we find that none specifically addresses crisis management systems or takes into account the 3D environment in which emergency situations occur. We have found that "workflows are used to support information dispatching within disasters" [4]. The proposed DSL applies model-driven requirements engineering to crisis management, hence providing built-in support for modeling location-based, social media-inspired interactions in the 3D environment of a crisis. While other aspects of disaster relief such as security and privacy are also critical for crisis management systems, we focus on the core elements involved in managing crisis events to build our DSL.

While traditional ways of contacting authorities have worked in the past, they limit the efficiency of crisis management. Indeed, the one-on-one voice only 911 phone calls to the police department, which may still be accomplished in parallel to using the mobile application we build as a proof-of-concept implementation, are time-consuming and may lead to misunderstandings [4]. Today, the role of social media in everyday life cannot be ignored. Social media has provided us with means to effectively collect valuable information from users all over the world [16]. Modern social media like interactions may thus be used for crisis management purposes, to speed up the crisis respondents' decision making process, help victims, and inform authorities of the emergency situation [16]. Hence, by using ad-hoc networks for victims and first responders to communicate, the aforementioned mobile application supports social media-inspired interactions, thereby facilitating the overall crisis management.

As emergency situations can occur anywhere around the world, crisis management systems are of great importance for everyone. The long-term goal is to reduce the damages or losses, in terms of property and life, caused by a crisis, benefiting us all as well as the environment we live in.

1.3 Methodology

To create our DSL for crisis management, we start by researching various types of emergency situations and decide to focus our attention on natural disasters, which are normally widespread, have severe consequences, and require urgent response. Then, we narrow down our search to three such crisis incidents, namely earthquakes, floods, and fires, and find resources comprising the steps to follow when dealing with each of these disasters. From papers and websites mostly written by researchers and community authorities, we write use cases per chosen emergency situation, in order to better understand the workflows needed to handle such events. Afterwards, we create a generic use case encompassing the steps common between the use cases of the three natural disasters, in order to obtain a more general set of steps applicable to the management of any crisis situation. Finally, we derive a list of requirements for the DSL from the written use cases and build a DSL for crisis management, covering the established requirements and based on the UCM workflow metamodel.

In our vision, crisis experts author emergency procedures by creating UCMs textually using the crisis management DSL. These UCMs then form the basis of a proof-of-concept implementation of a crisis management mobile application for victims and first responders, which we develop to validate our work. In fact, the traversal algorithm we design runs on an instance model of the TUCM to visualize the steps to follow when dealing with emergency situations on the mobile application. The envisioned mobile application is developed for the Android platform and used during emergencies to guide users on the actions to take. Some crisis scenario UCMs are devised and executed on the mobile application as means of testing and validating the feasibility and implementation of our crisis management DSL, and the correct functionality of our traversal mechanism. Note that we do not claim ourselves as being crisis experts and that the use cases included in this thesis are produced by the author after conducting research and thus may not be perfectly accurate. However, to the best of our knowledge, the use cases encompass the most essential information. Nevertheless, this does not affect the results obtained from

running the test cases as they are simply used to demonstrate the feasibility of our DSL and the correctness of our traversal algorithm.

1.4 Abbreviations

The abbreviations below are used throughout this thesis.

- BPD: Business Process Diagram
- BPMN: Business Process Model and Notation
- CEML: Crisis and Emergency Modeling Language
- DSL: Domain-Specific Language
- EMF: Eclipse Modeling Framework
- GPL: General-Purpose Language
- GRL: Goal-oriented Requirement Language
- ITU: International Telecommunication Union
- OMG: Object Management Group
- TUCM: Textual Use Case Map
- TURN: Textual User Requirements Notation
- UCM: Use Case Map
- UML: Unified Modeling Language
- URN: User Requirements Notation
- WS-BPEL: Web Services - Business Process Execution Language
- XMI: XML Metadata Interchange
- XML: eXtensible Markup Language

1.5 Thesis Organization

The remainder of this thesis is structured as follows:

Chapter 2 provides background details on crisis management systems, DSLs, workflow notations, and the differences between TURN and URN.

Chapter 3 presents the requirements we determine for a crisis management DSL and our proposed crisis management DSL based on those requirements.

Chapter 4 discusses the traversal mechanism we design to step through the textual crisis management UCMs created using the DSL.

Chapter 5 describes the application of the DSL for crisis management via our developed Android mobile application.

Chapter 6 validates the feasibility of our DSL and the correctness of our traversal algorithm, by demonstrating their functionality using the developed Android mobile application through testing selected UCMs reflecting real life crisis scenarios.

Chapter 7 reports related work, describing previous research involving the extension of existing workflow notations and their application, as well as the execution of existing scenario languages.

Chapter 8 concludes the thesis and discusses future work.

Chapter 2

Background

This chapter highlights background information on crisis management systems, domain-specific software languages (DSLs), workflow notations, and the differences between the User Requirements Notation (URN) [6] and its textual counterpart (TURN) [1], useful to understanding future chapters.

2.1 Crisis Management Systems

Crisis management aims to determine, evaluate, and handle emergency situations [17], which include natural disasters, attacks, accidents, epidemics, technological disruptions, and medical emergencies [18]. Such incidents are known to be unpredictable, thereby requiring proper means of dealing with them when they occur, particularly to mitigate their consequences on the environment, property, and life. Over the years, the demand for faster and more effective ways of responding to crisis events, in order to reduce the severity of their after-effects, has increased [18]. Crisis management systems were thus proposed to achieve this purpose by managing the communication between all individuals involved in a crisis and the steps to follow to quickly respond to the occurrence of an emergency situation [18]. As a matter of fact, during a crisis event, making decisions, processing crisis information, and communicating with other parties in a timely manner greatly affect the efficiency in managing crises, which is the problem crisis management systems are meant to solve [3, 17].

2.2 Domain-Specific Software Languages

Two approaches to defining a software language exist, namely generic and domain-specific software languages [19]. The former solves problems belonging to a number of domains, thereby making it less suitable for solving problems specific to a given domain, which is what domain-specific software languages strive to do [19]. In this thesis, we focus on the latter.

A domain-specific software language (DSL) focuses on one particular domain and is usually small [19, 20]. However, sometimes, in addition to the domain-specific concepts, the language may comprise all concepts of a general-purpose software language (GPL), such as the Business Process Model and Notation (BPMN) [5] or Unified Modeling Language (UML) [2] [19], where all generic and domain-specific concepts are contained within a metamodel on which the language is based [20]. The latter, representing an alternative to the costly development of a DSL from scratch independent from an existing language, implements the DSL by extending a basic GPL [9, 19], thereby reusing and specializing it [20]. Extending a language for a specific domain and for describing a DSL is a common practice [20]. A metamodel is a language design framework and visual representation of a workflow language, encompassing the basic workflow components, the relationships between them, their attributes, and their semantics [21–23]. In our case, we focus our attention on extending a basic workflow language (i.e., Use Case Map (UCM) [6]) with domain-specific concepts by adding these to the metamodel of the workflow notation. In order to develop a DSL, one must analyze the domain by determining the problem domain, assembling the most important information on the domain, grouping the concepts extracted from the found knowledge into a metamodel, and finally designing the DSL that describes the domain [19]. Numerous DSLs based on workflow notations exist today representing various domains such as aspect modeling, quality management, and performance [9], but - to the best of our knowledge - none of them focus on a workflow notation that considers the 3D environment in which emergency situations unfold and social media like interactions.

2.3 Workflow Notations

Many workflow notations exist and have been standardized by either the Object Management Group (OMG) [24] or the International Telecommunication Union (ITU) [25]. Popular notations are the OMG's Unified Modeling Language (UML) [2] Activity, Class, and Interaction Diagrams, or Business Process Model and Notation (BPMN) [5], and the ITU's User Requirements Notation (URN) [6] Goal Models and Use Case Maps. BPMN and UML are regarded as the main standards for business process modeling among notations and languages [12]. UML Activity Diagrams depict the activities that must be related to execute a UML use case [26], thereby modeling the control flow of process activities [22]. BPMN targets the design, execution, management, and analysis of business processes [27], which are collections of procedures seeking to achieve a business goal [4, 7, 12, 22]. BPMN represents business processes via its graphical notation, Business Process Diagram (BPD) [5, 7]. Similarly, URN assists in developing, describing, and analyzing requirements using the two notations it comprises [28]. On one hand, we have the Goal-oriented Requirement Language (GRL) [6], which helps in describing business goals, alternative solutions, non-functional requirements, and the reasoning behind them [28]. On the other hand, we find Use Case Maps (UCMs) [6], which describe functional requirements using maps of scenarios containing causally ordered responsibilities to fulfill these requirements [28]. As a result, URN captures needed functional and non-functional requirements via visual representations [28].

In this thesis, we focus on extending URN's UCMs (see example in Fig. 1), but the discussed crisis management concepts could also be introduced to other workflow notations. The UCM [13] workflow notation illustrates the causal interaction of entities. A *map* encompasses *paths*, which comprise path nodes. Paths start at *start points* (●, e.g., start online shopping) and end at *end points* (▮, e.g., delivered), with labels denoting the circumstance that provoked the sequence of steps and the outcome of a chain of steps, respectively. Steps to accomplish on a path are depicted by *responsibilities* (×, e.g., ship order). Furthermore, alternatives are represented by *OR-forks* (⌒╲), which

normally have guard conditions between square brackets on each branch, and *OR-joins* (⟍⟋), while concurrency is illustrated by *AND-forks* (⊣⊢) and *AND-joins* (⫤⊢). Note that any combination of forks and joins is allowed, having a join after a fork is not mandatory, and only one alternative path may be taken at a time. Paths can also contain *static stubs* (◇, e.g., make payment), which refer to at most one sub-map, named *plug-in map*. A path that reaches a stub enters the plug-in map it contains at its start point, continues until its end point, and moves back to the parent map to proceed. The connections between a stub and its plug-in map are formally defined with plug-in bindings. Fig. 1b shows the plug-in map of the static stub in the UCM in Fig. 1a. Other types of stubs exist and the interested reader is referred to the URN standard [6]. Moreover, map elements may be bound to *components* (□, e.g., WebStore), which indicate structural system elements. People or things interacting with the system are depicted by components of type *actor* (🯅, e.g., Customer). From these maps, various scenario paths may be simulated, yielding a test suite for scenario-based

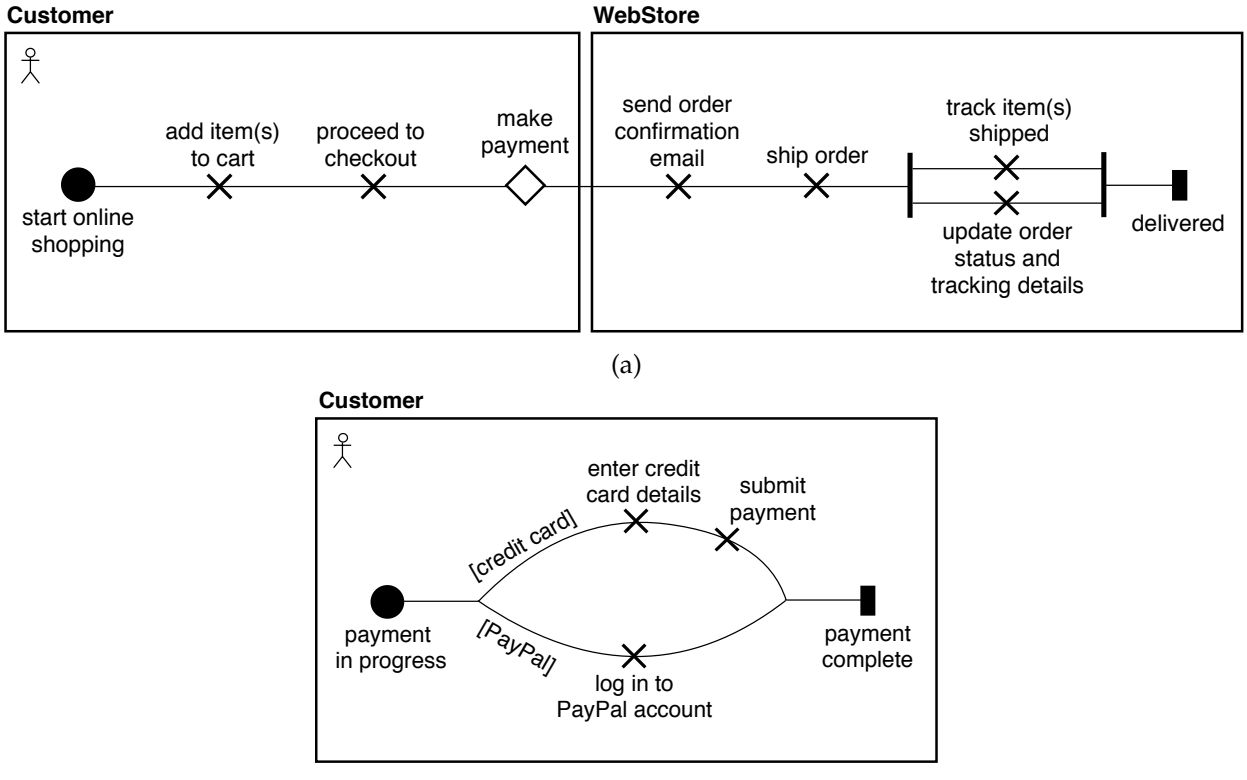


Figure 1: Example UCM in (a) with its "make payment" static stub's plug-in map in (b)

requirements with the help of UCM *scenario definitions*.

2.4 TURN vs URN

URN represents GRLs and UCMs graphically, while TURN, which is defined by the latest version of the URN standard [6], depicts these notations textually, thereby facilitating the definition and analysis of several distinct and complex GRL and UCM models, which would otherwise be harder to navigate. Moreover, inputting vast specifications graphically using URN is often deemed time-consuming and inefficient due to the need to carefully consider layout matters apart from modeling the relevant information. Therefore, TURN, the textual syntax for URN, provides modelers with an alternative mean of entering URN specifications which is simpler and more convenient, particularly in situations where large specifications are required. While TURN strives to support the most possible concepts of URN, it does not support some minor features of URN in a trade-off with practicalities of textual languages. Consequently, a model created via this textual notation is unable to describe everything that a model created via URN can specify. However, for what we want to achieve in this thesis, the URN concepts covered by TURN are sufficient to demonstrate our work. We base our work on an implementation of TURN using Xtext [1].

As this thesis only deals with UCMs, we will solely focus on comparing the URN and TURN specifications of the UCM notation, especially with regards to the notions relevant to our work, and omit mentioning these standards' specifications of the GRL notation. The Textual Use Case Map (TUCM) notation differs from its graphical UCM counterpart in a number of ways [1, 6]. In fact, component types present in URN are not supported in TURN, and URN's UCM Component and Responsibility concepts are covered in TUCM implicitly by defining the name and longName of ComponentRef and RespRef, respectively, as shown in Fig. 2. However, the main difference between the UCM and TUCM notations lies in their respective depiction of paths. As illustrated in Fig. 2, in UCMs, paths can be shaped differently using empty points, which make the latter irrelevant textually, in TUCMs.

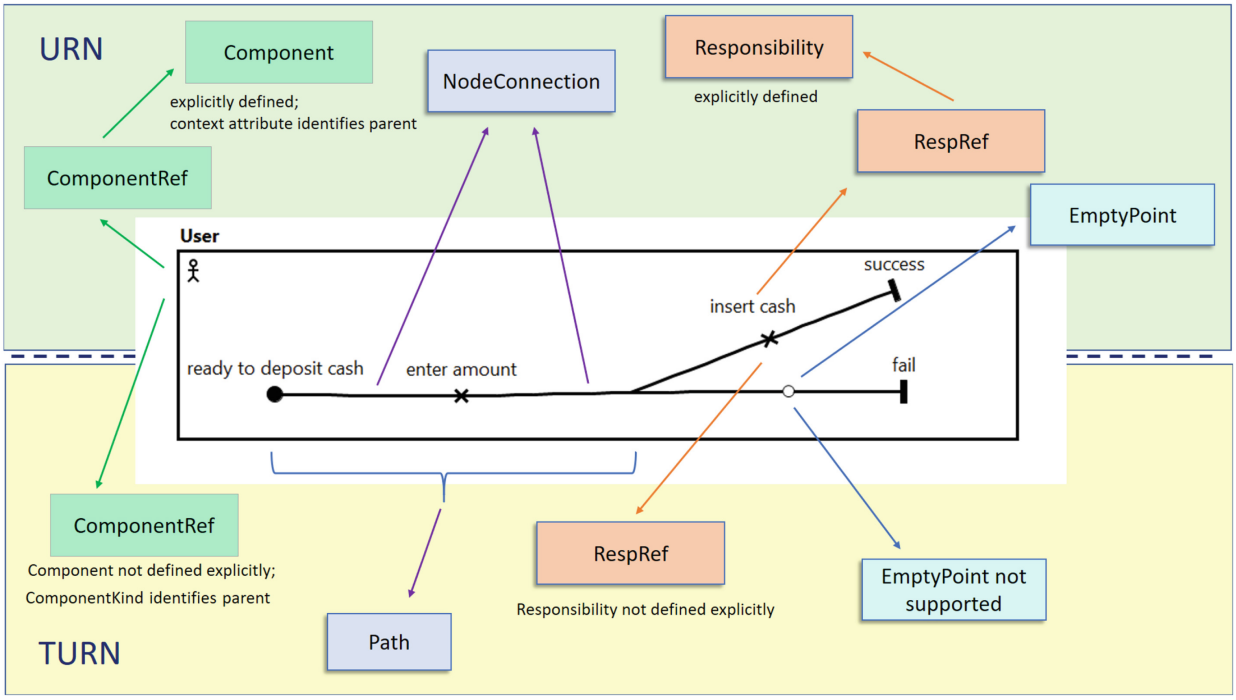


Figure 2: Overview of the main differences between UCM and TUCM [1]

A more notable distinction between the two notations' representations of paths involves the existence of the Path concept in TUCMs, which is not explicitly specified in UCMs. Rather than using NodeConnection to connect consecutive path nodes as in a UCM, a TUCM implicitly links path nodes by ordering them according to when they occur on a Path. A Path in a TUCM begins with a StartPoint and is made up of PathBody, which are segments, or branches, of a UCM found between branching points, i.e., forks, joins, or stubs. PathBodyNodes lie on a PathBody. Fig. 3 portrays these PathBodyNodes, which are equivalent to URN PathNodes. A PathBody may end with a RegularEnd, which corresponds to an EndPoint or any path element where new PathBodies begin, i.e., AndFork, OrFork, or Stub, or with a ReferencedEnd referring to a previously specified path element, i.e., AndJoin, OrJoin, or Stub. In other words, a TUCM replaces the PathNode and NodeConnection concepts in a UCM by the notions of Path, PathBody, PathWithRegularEnd, PathWithReferencedEnd, PathWithReferencedStub, PathBodyNodes, PathBodyNode, RegularEnd, and ReferencedEnd. The aforementioned TUCM notions are used to break down a UCM into separate segments, thereby allowing the definiton of UCMs textually in a tree-based

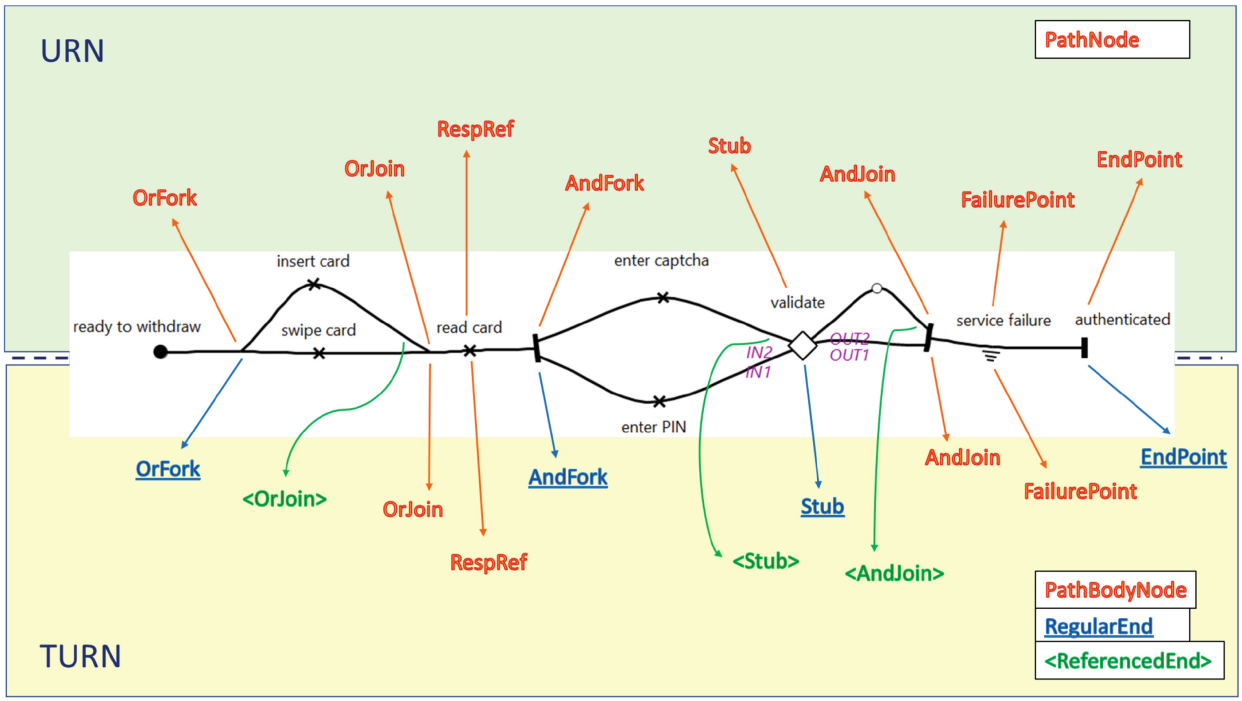


Figure 3: Differences in depicting paths between UCM and TUCM [1]

structure.

In addition, OrJoins and AndJoins may be implicit in TUCM models using the shortcut notation which consists in ending the branches of OrForks and AndForks with a semi-colon before the connecting PathBody (see TUCM in Listing 1 and its corresponding UCM in Fig. 4), while OrJoins and AndJoins are explicit in UCM models, as demonstrated in Fig. 5 for the case of OrJoins.

Listing 1:
Example TUCM containing Implicit OrJoin and Implicit AndJoin

```
urnModel JoinsExample

map OrForkAndForkJoinsExample {
start s
  -> or { [condition1] -> X r1 -> ;
           [condition2] -> X r2 -> ;
     }
  -> and { * -> X r3 -> ;
           * -> X r4 -> ;
     }
  -> end e.
}
```

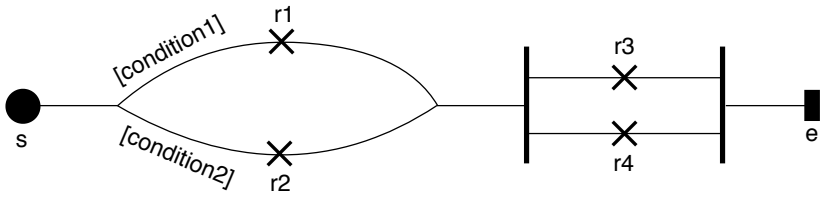


Figure 4: UCM corresponding to the TUCM in Listing 1

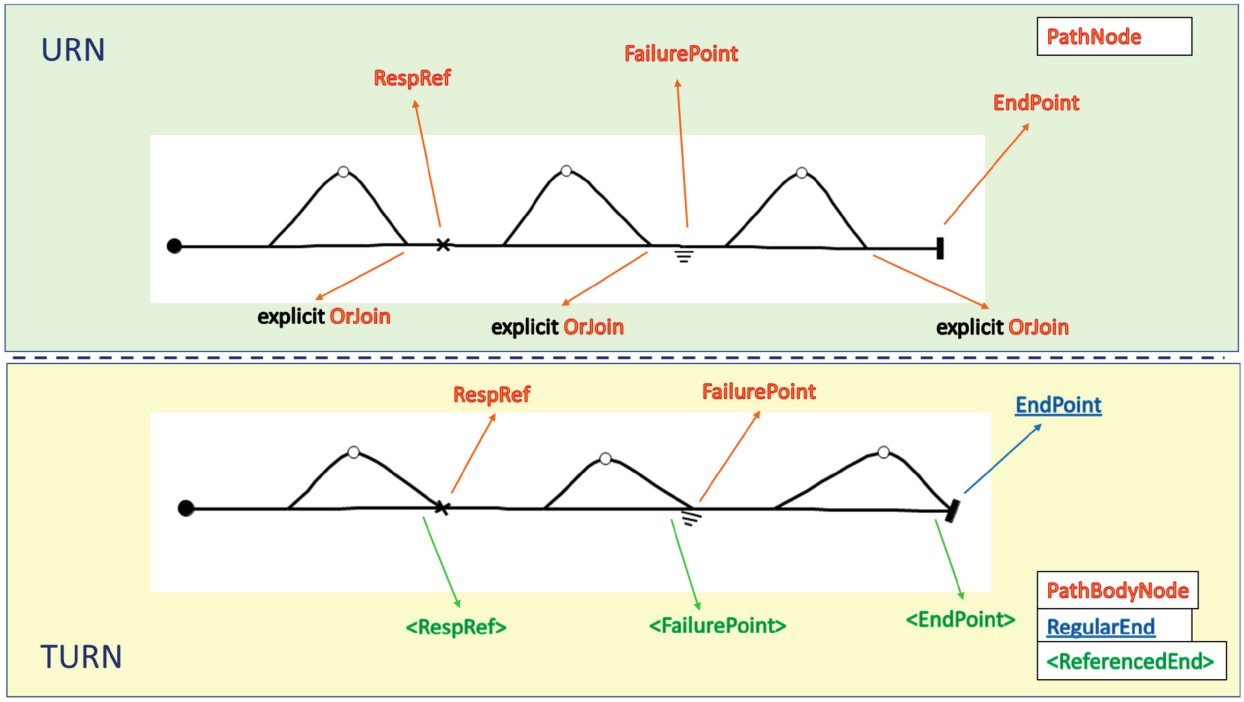


Figure 5: Differences in representing OrJoins between UCM and TUCM [1]

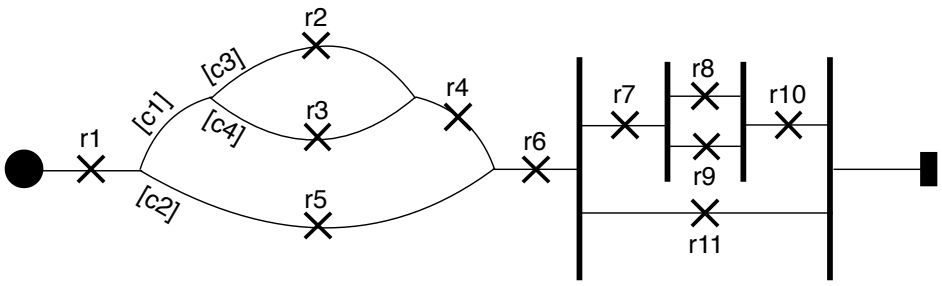


Figure 6: Example UCM with two ConnectingOrBodies, one containing path node r4 and another containing path node r6, and two ConnectingAndBodies, one containing path node r10 and another being empty

Furthermore, OrFork/OrJoin and AndFork/AndJoin pairs are followed by ConnectingOrBodies and ConnectingAndBodies respectively, as shown in Fig. 6. In Fig. 6, for the case of ConnectingOrBodies, the OrFork/OrJoin pair with branch conditions c3 and c4 is followed by the ConnectingOrBody containing path node r4, while the OrFork/OrJoin pair with branch conditions c1 and c2 is followed by the ConnectingOrBody containing path node r6. On the other hand, for the ConnectingAndBodies in Fig. 6, we have the ConnectingAndBody containing path node r10 following the AndFork/AndJoin pair consisting of path nodes r8 and r9, while an empty ConnectingAndBody follows the outermost AndFork/AndJoin pair consisting of path nodes r7, r8, r9, r10, and r11.

2.5 Summary

The background details regarding crisis management systems, DSLs, workflow notations, TURN, and URN explored in this chapter are of fundamental importance in understanding the chapters hereinafter as they build on the aforementioned concepts. In fact, the following chapter presents our proposed crisis management DSL, extending the URN UCM workflow notation.

Chapter 3

DSL for Crisis Management

The approach to take to gather requirements for and create our crisis management domain-specific software language (DSL) is described in this chapter.

3.1 Requirements for a Crisis Management DSL

In this section, we present the process to follow to determine the requirements for a crisis management DSL. We describe the conducted preliminary research and the use cases detailing the steps to follow for chosen emergency situations.

3.1.1 Preliminary Research

To realize the goal of our research, we identify crisis management as our problem domain and gather knowledge regarding this domain, as is required in order to create a DSL [19]. The latter is accomplished by first looking up various emergency situations online, which we narrow down to handling natural disasters, since these events require more immediate responses as they tend to have consequences of greater extent. As the list of natural disasters can be quite lengthy, we focus on three of them: earthquakes [29, 30], fires [31, 32], and floods [33, 34]. For the chosen crisis situations, we research websites and articles detailing the steps to deal with them, including tips on how to be better prepared for the eventuality of such disasters. Many of the resources used are written by communities following predefined protocols, thereby making them more reliable in determining crisis management concepts applicable to similar situations.

3.1.2 Use Cases

We employ use cases to determine a system's functional requirements as they facilitate the creation of the system's domain model [26]. For each of the three chosen emergency situations, to solidify our understanding, the author wrote key use cases, using the steps found in the emergency plans of various communities around the world, which were reviewed by the author's supervisor. Then, we determine the steps that are common between the use cases of each disaster and use them to write a generic use case encompassing more general steps to manage any crisis, shown in Listing 2.

From the set of use cases and the situations common between the three kinds of natural disasters considered, we gather a list of requirements for a crisis management DSL. First, since actors, i.e., victims and authorities, are involved in each emergency situation, we determine the need for a clear way to distinguish their individual set of actions during crises using the DSL. Second, we notice that various resources are needed in all cases to handle the disaster and thus conclude that indicating necessary resources, and whether they can be found at the crisis location or obtained from authorities, must be covered by the DSL. Third, in the use cases of all investigated crisis incidents, victims and authorities have to communicate with each other to report the disaster, and send the required emergency personnel and resources to the location of the disaster. Thus, the DSL must be able to model this communication between actors in dynamically changing groups. Fourth, some of the use cases comprise actions that victims or authorities must abide to and warn others of for safety reasons. Hence, warnings must be covered by the DSL. Fifth, some use case steps encompass reasons why a given action must be taken. In other words, the DSL must allow the indication of the rationale behind some actions to be taken. Last, the DSL must be capable of denoting the location of emergencies, and their associated actors and resources.

Therefore, a DSL for crisis management must:

(R1) clearly distinguish the actors involved in a disaster and their respective set of actions for handling the disaster

(R2) indicate resources, including whether they are found at the location of a crisis or

Listing 2:
Generic use case applicable to any crisis situation

Use Case: Handle crisis situation

Primary Actor: Victim

Secondary Actor: Authorities

Precondition: Crisis situation has occurred.

Main Scenario:

1. Victim alerts anyone in immediate vicinity so that they also warn the people around them, and ultimately everyone is warned, but reminds them to stay calm.
2. Victim relocates to a safe area, and avoids risk areas.
3. Victim contacts Authorities to inform them of the crisis situation and its possible damages in terms of life and property.
4. Victim remains where he or she is until further instructions are provided by the Authorities.

Step 5 done in parallel with all remaining steps

5. Victim monitors communication with Authorities.
6. Victim waits for Authorities to arrive, if they are to come, or to announce when it is safe to resume regular activities.
7. Victim checks whether surrounding victims require medical attention.
8. Victim provides assistance to other victims, if doing so does not put him or her in danger.
9. Victim attends to other victims medically only if he or she has the training to do so, has called for medical help, and the victim to attend to is highly injured and requires immediate assistance which cannot wait until the paramedics arrive.
10. If Authorities announce the need for an evacuation, Victim assembles other victims, manages the flow of evacuees, and guides the evacuation process to a space away from the crisis situation.
11. Victim prevents re-entry to crisis area.
12. Once Authorities have announced that it is safe, Victim goes back to what he or she was doing prior to the occurrence of the crisis situation.

Postcondition: Crisis situation has been handled.

obtained from authorities

(R3) model the communication between groups of actors

(R4) indicate critical actions that victims and authorities must be aware of

(R5) depict the reasoning behind taking some actions

(R6) indicate the location of emergencies, and their associated actors and resources

3.2 Proposed DSL for Crisis Management

First, we consider the Use Case Map (UCM) [6] metamodel, which we extend with crisis management concepts determined from the requirements established in the previous section. Then, for visualization purposes, we explore the changes we make to the concrete syntax of UCMs in order to capture the concepts required to manage crises. While the crisis management concepts added to UCM apply to its graphical representation, we internally use the textual representation of UCMs based on the Textual User Requirements Notation (TURN) [1] to build our crisis management DSL and facilitate our proof-of-concept implementation.

3.2.1 Metamodel

The crisis metamodel is separated into three views to facilitate its understanding, as demonstrated in Fig. 7 depicting the overall structure of the metamodel and the associations between these different views. Figs. 8, 9, and 10 correspond to these three views, and show the metamodel we create from the crisis management concepts determined from the aforementioned use cases and how these elements are associated to current and added classes of the User Requirements Notation's (URN) [6] UCM metamodel. The execution view in Fig. 8 captures the domain concepts of crisis management systems, i.e., its instances describe the actual real life entities of a given emergency. The emergency is handled according to a plan defined by the type view in Fig. 9. Each concept in the execution view corresponds to one concept in the type view according to the abstraction (type)/occurrence (execution) pattern as shown by the mappings in Fig. 10.

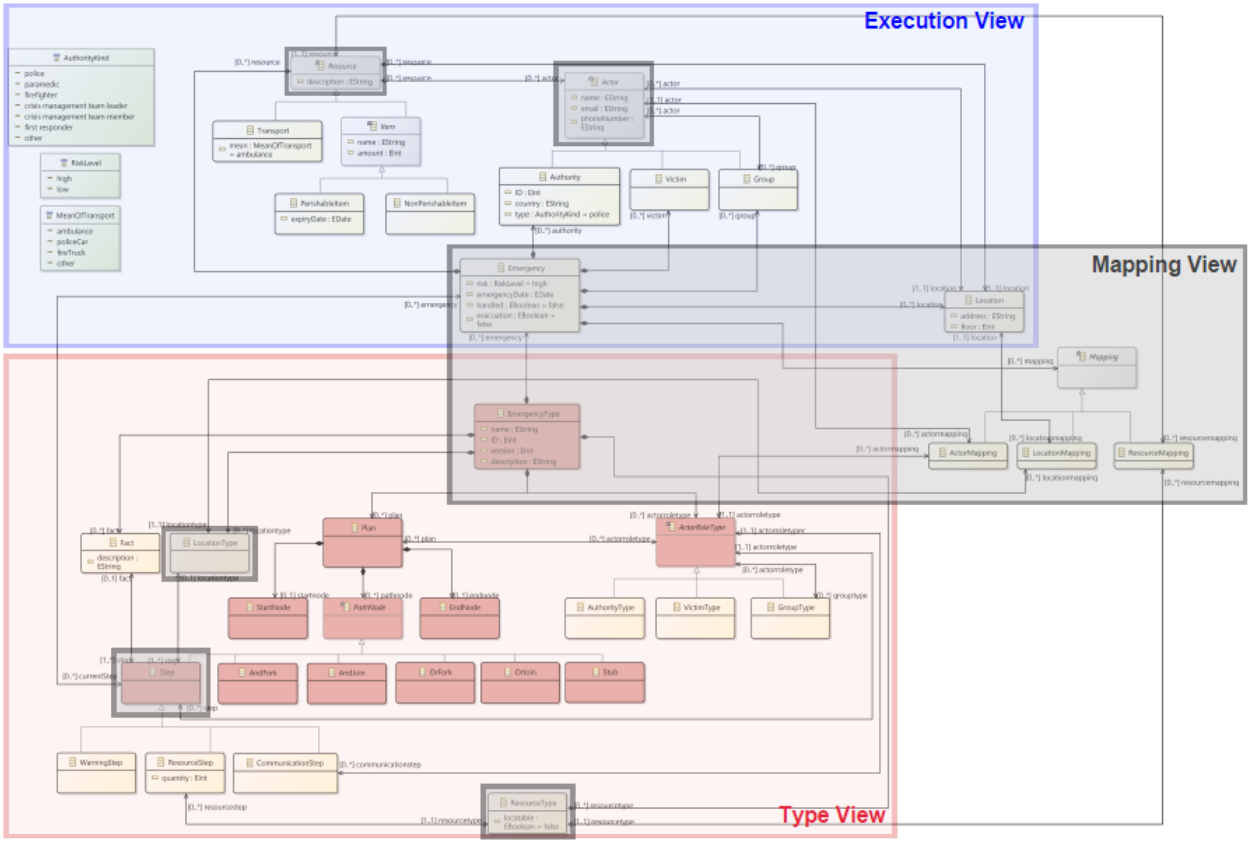


Figure 7: Overview of the crisis metamodel showing the associations between its three views, execution (in blue), type (in red), and mapping (in grey)

3.2.1.1 Crisis Management Concepts

These aspects of crisis management (see Fig. 8) primarily focus on an Emergency that is currently active, i.e., it is executing, and comprise the Emergency that occurs, with its attributes, including the risk level of the disaster, since situations of low risk can be handled by the victim himself or herself, and booleans for whether an evacuation is required and whether the crisis has been handled, determined from the written use cases. Moreover, we incorporate the concept of individuals interacting with each other during an emergency, which consist of the Victim experiencing a crisis and the Authority, such as a firefighter, policeman or paramedic, responding to the crisis. These notions are determined by looking at the primary and secondary actors specified in the use cases of the chosen disasters. In addition to the Authority and Victim subtypes of the Actor class, the subtype Group handles teams of authority figures working together, groups of victims, and a combination

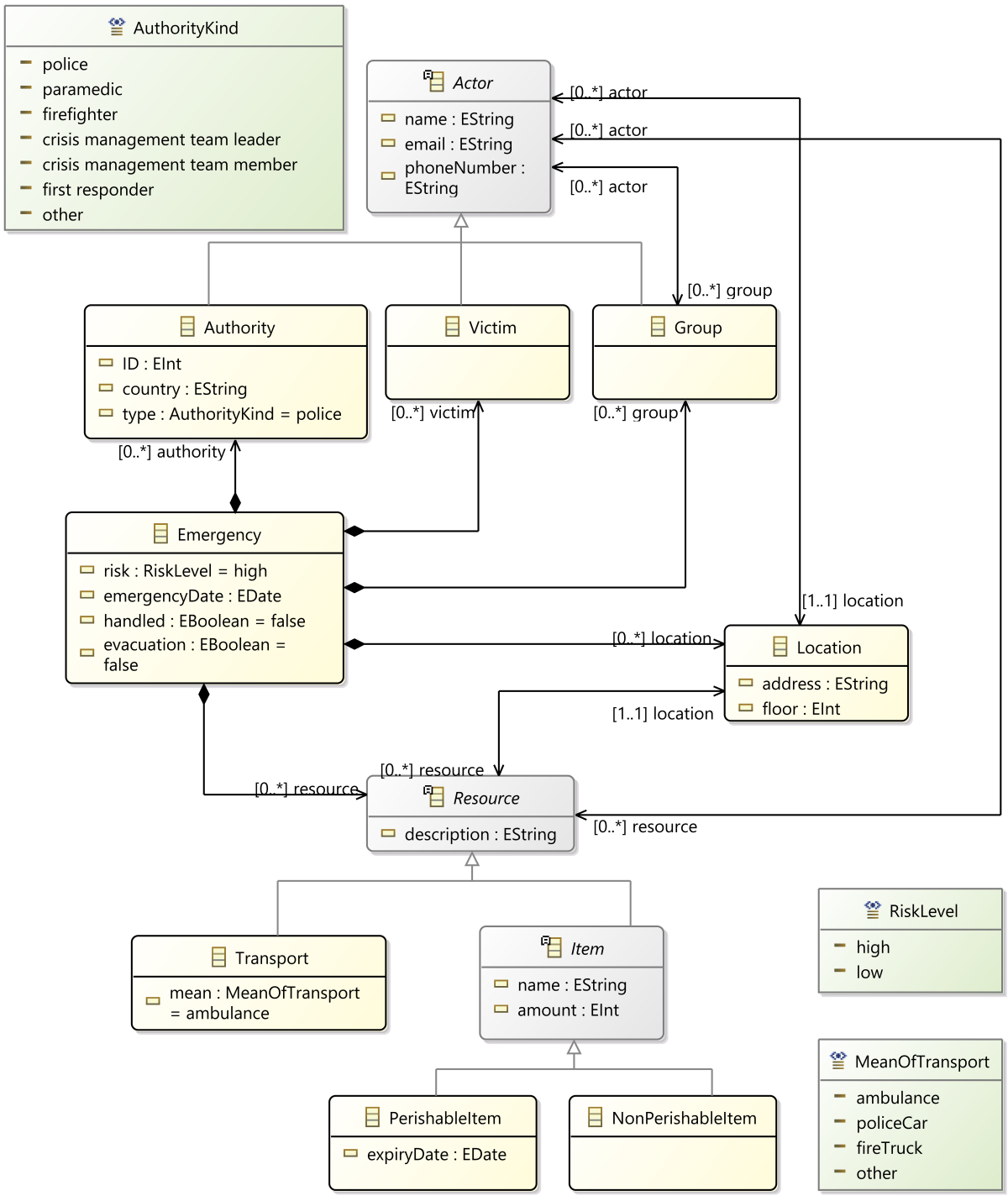


Figure 8: Execution view of the crisis metamodel

of both victims and authorities. In some of the use cases of the chosen emergency situations, actors require special Resources, either PerishableItem or NonPerishableItem, or Transport vehicles, to cope with the disaster. Such resources include fire alarms, fire extinguishers, defibrillators, and first aid kits, usually found near the location of the crisis incident, and firetrucks, police cars, and ambulances, as needed. From existing descriptions of crisis management situations, we determine that authority figures also use nonperishable items, such as drones to locate victims in widespread crisis events or to determine the extent of a disaster. In the metamodel, the Location class captures the information regarding where an emergency situation occurs, and where actors and resources are located. Lastly, the ad-hoc network of victims impacted by a given crisis incident can be determined through their location, when compared against the location of a reported disaster and the location of other victims, to support social media-inspired interactions.

3.2.1.2 Extended UCM Metamodel Concepts

The aforementioned notions regarding crisis management are then mapped to existing and added concepts of UCMs, as depicted in Fig. 9. In Fig. 9, classes in red correspond to crisis management concepts mapped to existing URN UCM metamodel ones, with the name of their equivalent UCM concept clearly denoted, if the names differ, while all other classes represent the concepts we add to the UCM metamodel. This allows an EmergencyType to be defined by crisis experts, which subsequently is executed when an actual Emergency occurs. First, the UCM root, i.e., URNspec, is substituted with the EmergencyType class to represent that each UCM model is intended to handle a specific emergency type. The EmergencyType class is composed of the Plan class and ActorRoleType class (UCMmap and Component in the UCM metamodel, respectively), which serves the purpose of mapping an actual Actor to his or her role type during a given disaster, within the UCM. The Plan class is composed of the StartNode, EndNode, and PathNode, where the latter, as in the UCM metamodel, is the supertype of the OrFork, OrJoin, AndFork, AndJoin, and Stub classes. Note that there is additional complexity involved in the Stub concept, which is not shown in Fig. 9 as it is the same as in the

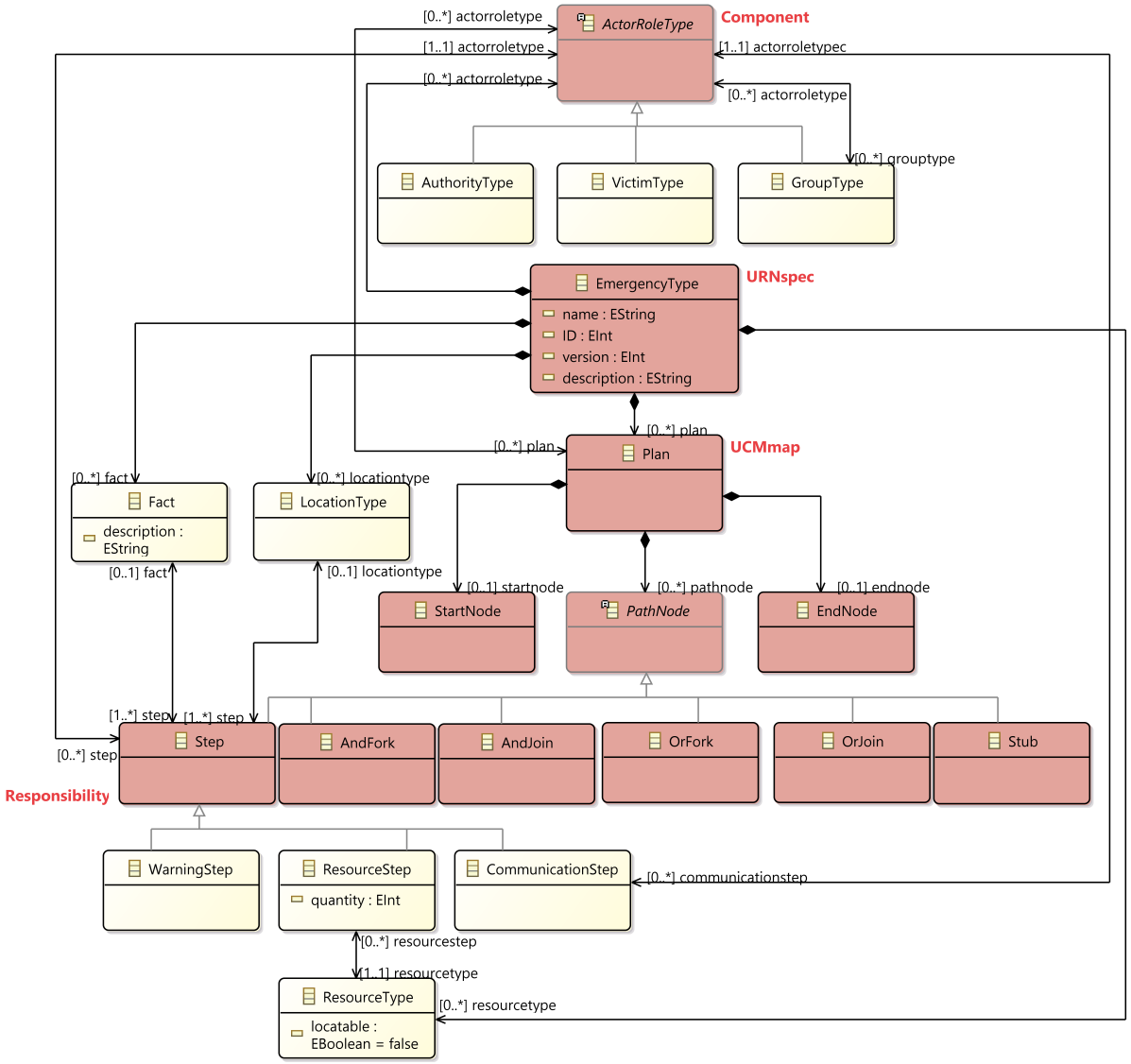


Figure 9: Type view of the crisis metamodel

UCM metamodel. In addition to the aforementioned subtypes, the Step (Responsibility in UCM metamodel) subtype specifically handles crisis management elements. Subtypes of the Step class include the WarningStep (to inform victims or authorities about a critical action they must abide to as the risks would be high otherwise), CommunicationStep (to represent a step in which victims or authorities must come into contact with other victims or authorities), and ResourceStep (to depict that a victim or authority requires a resource to handle the disaster). This last subtype is associated with the ResourceType class,

specifying whether a resource may be found by the victim or authority, indicated by the boolean locatable value true, or obtained by request from an authority, designated by the boolean locatable value false. Moreover, the CommunicationStep class is associated with the ActorRoleType class to depict the actors that communicate in a given step. A concept of crisis management added to the UCM metamodel is the Fact class, associated with the Step class, and used to provide important facts to a victim or authority regarding the reasons behind fulfilling certain actions in order to handle a given emergency situation. Another crisis management element also associated with the Step class is the LocationType class to keep track of the whereabouts of ActorRoleTypes performing a step or ResourceTypes associated with a ResourceStep.

Lastly, we need a way to track the step-by-step process of coping with a disaster by knowing the current step reached in the UCM by the victim or authority (see Fig. 10). The latter is taken care of by linking the Emergency class to the Step class, to know at which step a victim or authority performing the step is within an emergency situation. In general, all other type classes (ActorRoleType, ResourceType, and LocationType) need to be mapped to the actual Actor, Resource, and Location classes, respectively, when the

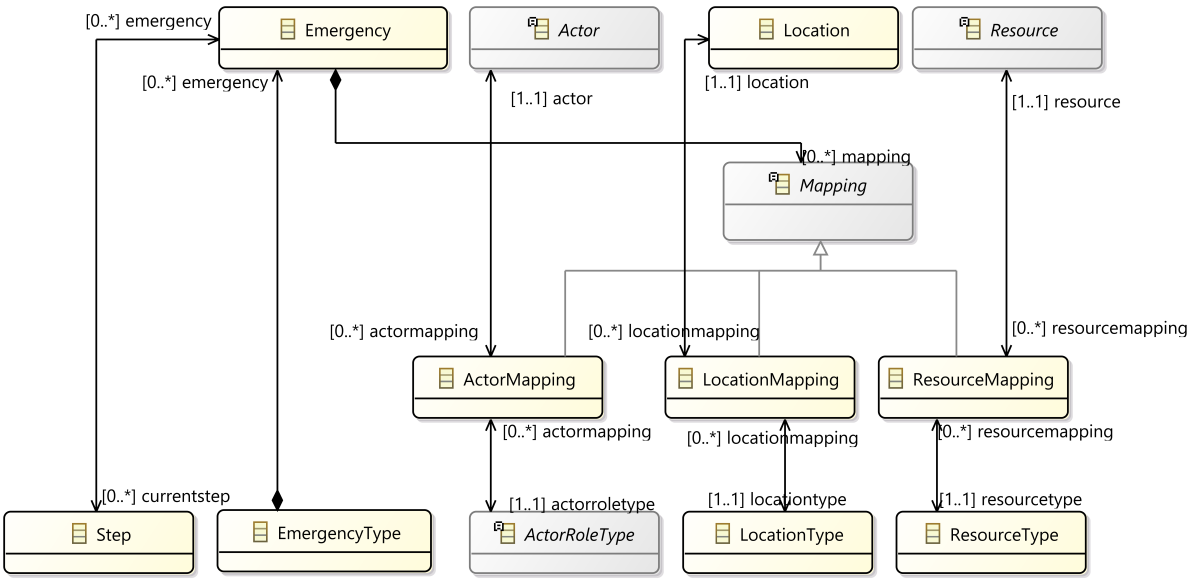


Figure 10: View of the mapping between the execution and type views of the crisis metamodel

Table 1:
Requirements and the metamodel concepts addressing them

Requirement	Metamodel Concepts
R1	ActorRoleType, VictimType, AuthorityType, GroupType, Actor, Victim, Authority, Group, ActorMapping, EmergencyType, Emergency, Plan, Step
R2	Resource, Item, PerishableItem, NonPerishableItem, Transport, ResourceType, ResourceStep, ResourceMapping
R3	CommunicationStep, ActorRoleType, Actor, ActorMapping, Location, LocationType, LocationMapping
R4	WarningStep
R5	Fact, Step
R6	Location, LocationType, LocationMapping, Step

actual Emergency related to a specific EmergencyType is executed. This is accomplished by the Mapping class and its subclasses.

With all the aforementioned concepts considered with respect to crisis management, we thus capture all relevant aspects present in the 3D environment in which a disaster occurs in support of social media-inspired interactions, thereby meeting the requirements listed in Section 3.1.2, as demonstrated in Table 1. While natural disasters are chosen as the main focus when conducting the preliminary research, once the generic use case is produced and the crisis management concepts are determined, we evaluate their applicability to other crisis situations, such as medical emergencies which are more common than, for instance, earthquakes, and conclude that these crisis events do not require additional concepts.

3.2.2 Use Case Maps

The six extensions we make to current UCM representations are shown in Fig. 11, which illustrates an example of procedure followed to handle a fire emergency from the viewpoint of a victim located at the site of the crisis situation. While Fig. 11 shows the additional perspectives of the police and fire departments, our proof-of-concept implementation assumes one role at the time of a crisis, in this case that of the victim. Handling a given type of emergency from one actor's perspective is represented using one extended UCM.

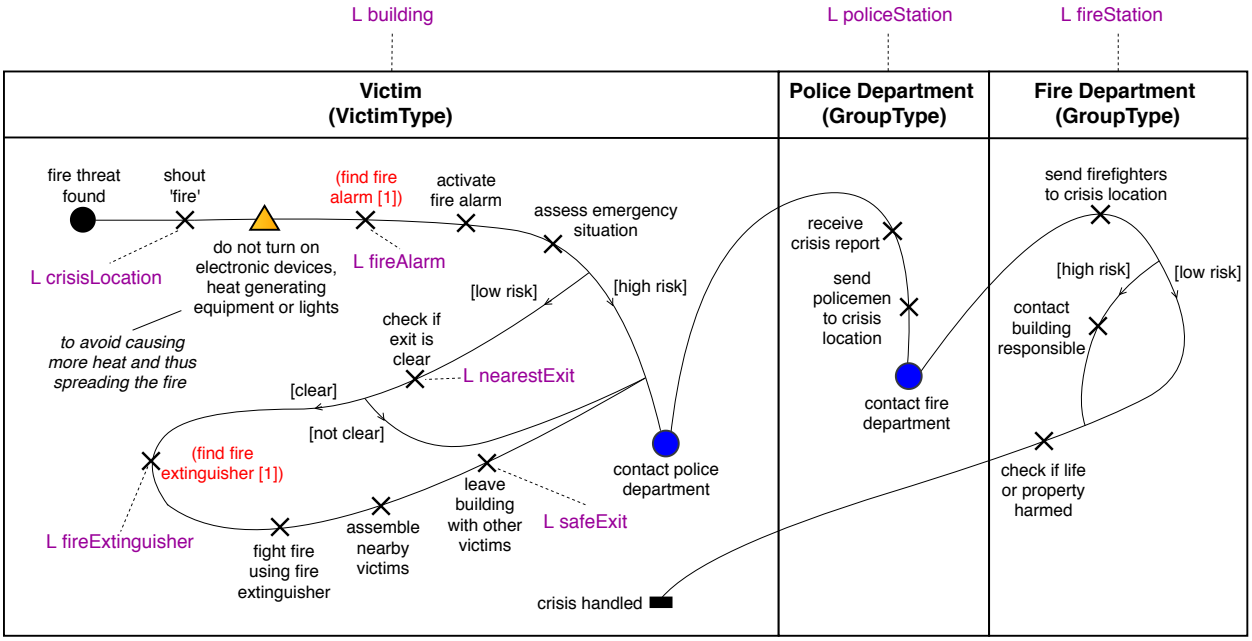


Figure 11: Example UCM for fire emergency

The UCM extensions are detailed as follows, with Fig. 12 clearly depicting their concrete syntax on the left and abstract syntax on the right.

ActorRoleType. As in activity diagrams, and instead of having the usual UCM components

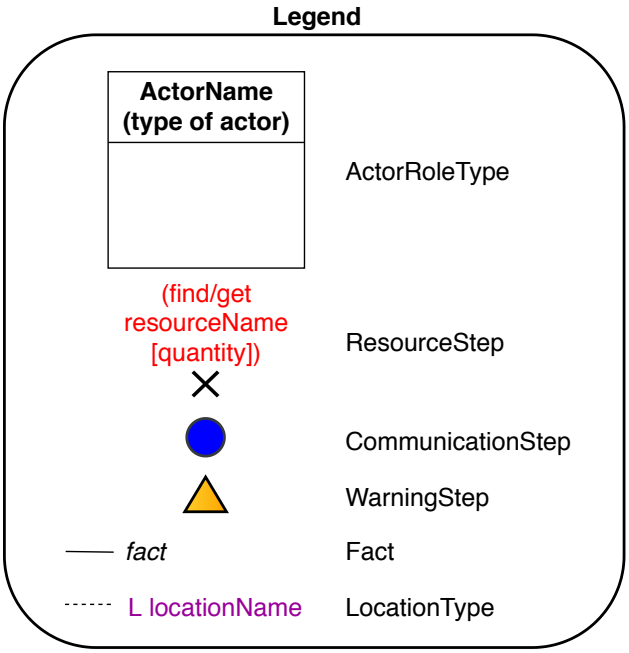


Figure 12: Concrete (on the left) and abstract (on the right) syntaxes of the extended UCM graphical notation

to depict actor roles, we represent all the steps handled by an actor, having a particular role, within his or her swimlane (i.e., a rectangular area, where all rectangular areas are the same height and placed next to each other). The type of actor, i.e., victim, authority, or group, is specified in parentheses below the role name of the actor.

ResourceStep. Resource steps are illustrated by labels in red font and surrounded by parentheses, where the text states whether the resource must be found by the victim or authority, denoted by the word "find", or obtained from an authority, specified by the word "get", followed by the name of the resource, and the quantity required of the resource between square brackets. A single resource step comprises a single resource type to acquire.

CommunicationStep. We add the blue circle symbol to depict the need for a victim or authority to contact another victim or authority at that step. Who needs to communicate is determined by the actor of the step and the actor mentioned in the label of the step.

WarningStep. Important warnings which victims or authorities should be aware of are represented by the orange triangle symbol and expressed in words through the label.

Fact. The notion of a "fact" is portrayed by a solid line directed out of a step's label with text in italic font describing the fact.

LocationType. The location of an emergency, resource, or actor as he or she progresses through an emergency plan is expressed using the capital letter L followed by a name designating what is being located, in purple font, and with a dotted line directed out of a step. We also connect the initial location of each actor to the top of its swimlane using the same format.

3.3 Summary

This chapter presents the preliminary research regarding natural disasters conducted to write use cases, which ultimately led to the deduction of a list of requirements for our crisis management DSL. These requirements are then used to establish the crisis management concepts which extend the UCM metamodel. Six extensions to the current UCM representation are thus determined and represented using its graphical notation for

illustration purposes. For this thesis, we extend TURN's Textual Use Case Map (TUCM) [1] notation with the concepts discovered in this chapter to build our crisis management DSL so that we can create textual models using our DSL. We then employ our traversal mechanism discussed in the next chapter to step through the emergency situation workflows described by our textual models in forward and backward direction.

Chapter 4

Traversal Mechanism for Crisis Management TUCM

Once Textual Use Case Map (TUCM) [1] models are created using our crisis management domain-specific software language (DSL), we run our traversal algorithm to allow the step-by-step forward and backward navigation of the produced emergency plans. The algorithm operates on a model instance of the Textual User Requirements Notation (TURN) [1] metamodel, and hence the textual representation is first converted into a model instance and saved as an XML Metadata Interchange (XMI) [35] file with the help of built-in Xtext features. Note that we assume that the model instance is well-nested and that its stubs are flattened prior to being fed to our algorithm to reduce the complexity of the traversal algorithm. This chapter explains our algorithm which consists in reading a specific crisis management TUCM in an XMI file, stepping through the Path in the selected model and, in the event that an AndFork is reached, using a separate data structure to traverse the AndFork until its end before continuing with the main data structure for the model's Path.

4.1 Important Considerations

While devising our algorithm, we make some significant choices. First, we decide to deal solely with well-nested models, to avoid complex and unique AndFork and OrFork branch hierarchies. Although TUCMs do not necessarily exert this property, we assume that well-nested models are generally easier to understand for a user following the steps in a crisis response scenario. In addition, we assume that stubs are flattened (as discussed in the URN standard [6]) before the instance models are fed to our algorithm, i.e., their

sub-maps are brought up to the highest level possible to make a model Path simpler to traverse. Moreover, whenever an AndFork is reached, a separate data structure is used to traverse the AndFork until its corresponding AndJoin, before continuing the steps on the Path of the model. We do this to have a clear separate representation of all parallel branches that have to be traversed and to isolate the complexity encountered while developing the part of the algorithm dealing with AndForks from the rest of the algorithm. Our traversal mechanism also handles iterating backward through a model's Path, including its possible AndForks. When navigating back across an AndJoin into an AndFork, the AndFork is reestablished, which means that it is retrieved at the state it was left at before continuing on the branch it lies on (i.e., the last step for each branch of the AndFork).

Another assumption we make involves the addition of an empty PathBodyNode, i.e., a blank regular Step, within AndForks on a TUCM's Path before OrForks and AndForks starting an AndFork branch and before an AndFork representing the first path element after the StartPoint on a TUCM's Path, if these elements are not already directly preceded by a step, prior to running our algorithm on an instance model. This case is illustrated in Fig. 13, where added empty PathBodyNodes are shown in red. This assumption is necessary for proper traversal in TUCMs with AndForks. Furthermore, we assume that AndForks in

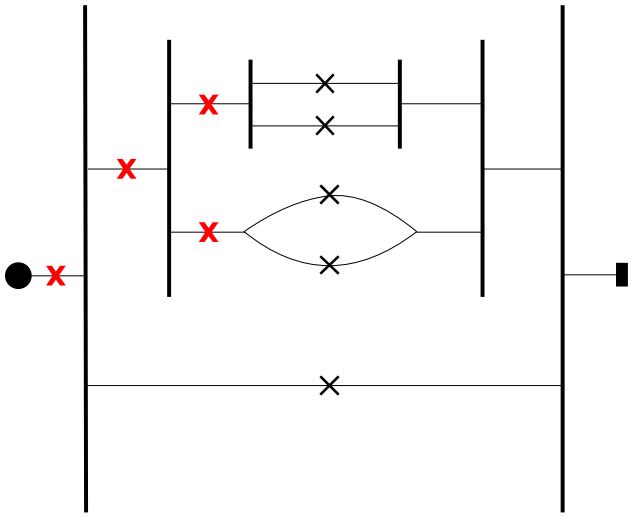


Figure 13: Addition of an empty PathBodyNode (in red) within AndForks before an OrFork or AndFork beginning an AndFork branch and before an AndFork representing the first path element after the StartPoint

the textual models are not empty. An empty AndFork branch serves no purpose as no concurrent action must be taken on it and thus we assume that it it is eliminated from a TUCM before running our algorithm on an instance model. However, an OrFork may contain a branch that is empty, which our algorithm supports, since we could have one alternative path requiring additional steps that do not apply to another alternative path, which would thus be empty (i.e., as in an if statement without an else body).

4.2 Traversal Data Structure

Our algorithm employs a doubly linked list to store in order the steps, or PathBodyNodes, that have been traversed, as this type of data structure allows each node in the list to contain a pointer to the node that precedes and follows it (see Fig. 14), thereby enabling the navigation of an emergency plan forward as well as backward. In fact, a workflow's state must be capable of capturing information regarding the steps already accomplished and those that are currently in progress, the paths taken at OrForks, etc. [8], which is what a doubly linked list essentially does by remembering the various states in which a TUCM is in. The doubly linked list also addresses a peculiarity of the TURN metamodel, where a node does not directly know its predecessors and successors.

We use one such data structure for a TUCM's Path but, upon reaching an AndFork, and any of its nested AndForks, we create a new doubly linked list for each AndFork branch to have a clear separate list for each parallel branch. We choose to employ a dedicated doubly linked list per AndFork branch to be able to manipulate each branch individually, which is useful in Chapter 5 when designing an Android mobile application where each

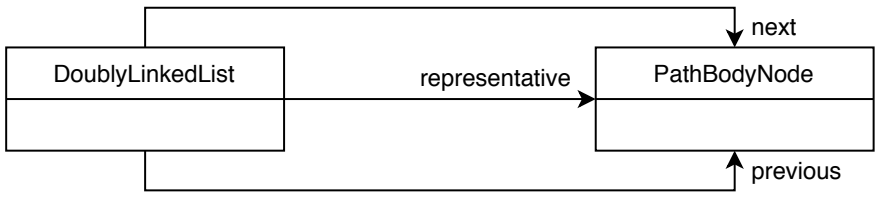
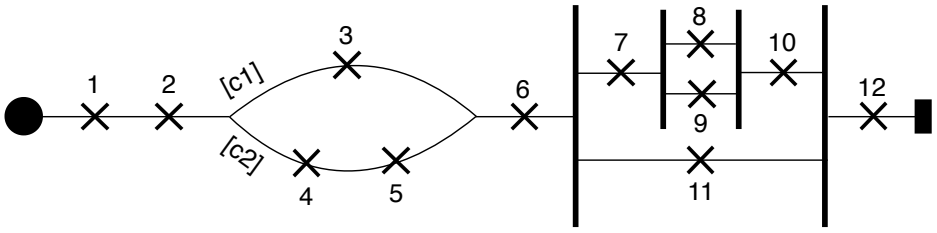


Figure 14: Unified Modeling Language (UML) [2] class diagram representing the doubly linked list data structure



2 Possible Doubly Linked Lists for Model Path Traversal:
(Main list) Choosing OrFork branch with condition c1: 1 ⇄ 2 ⇄ 3 ⇄ 6 ⇄ 7 ⇄ 12
(Main list) Choosing OrFork branch with condition c2: 1 ⇄ 2 ⇄ 4 ⇄ 5 ⇄ 6 ⇄ 7 ⇄ 12

4 Doubly Linked Lists for AndFork Traversal:
(1) Top AndFork branch (connected to 7 in main list): 7 ⇄ 8 ⇄ 10
(2) Bottom AndFork branch (connected to 7 in main list): 11
(3) Top nested AndFork branch (connected to 8 in list 1): 8
(4) Bottom nested AndFork branch (connected to 8 in list 1): 9

Figure 15: Example UCM with corresponding doubly linked list representation

concurrent branch is represented by its own button allowing the forward and backward traversal of the steps on the specific branch. When stepping into an AndFork, the first PathBodyNode on its first branch is added to the doubly linked list of the branch on which the AndFork lies as a representative PathBodyNode (see Fig. 14) allowing us to know whether an AndFork has been traversed and to reestablish the AndFork when navigating backward in a TUCM. As discussed in Fig. 13, such a PathBodyNode always exists in an AndFork that is being traversed.

Fig. 15 captures our use of the doubly linked list data structure in the aforementioned way by providing a Use Case Map (UCM) [6] example with its corresponding doubly linked list representation. The PathBodyNodes added to a doubly linked list as representative PathBodyNodes of their AndForks (i.e., the first PathBodyNode in the first branch in an AndFork) are PathBodyNode 7, which is part of the main doubly linked list of the model Path, and PathBodyNode 8, which is part of the doubly linked list of the top AndFork branch.

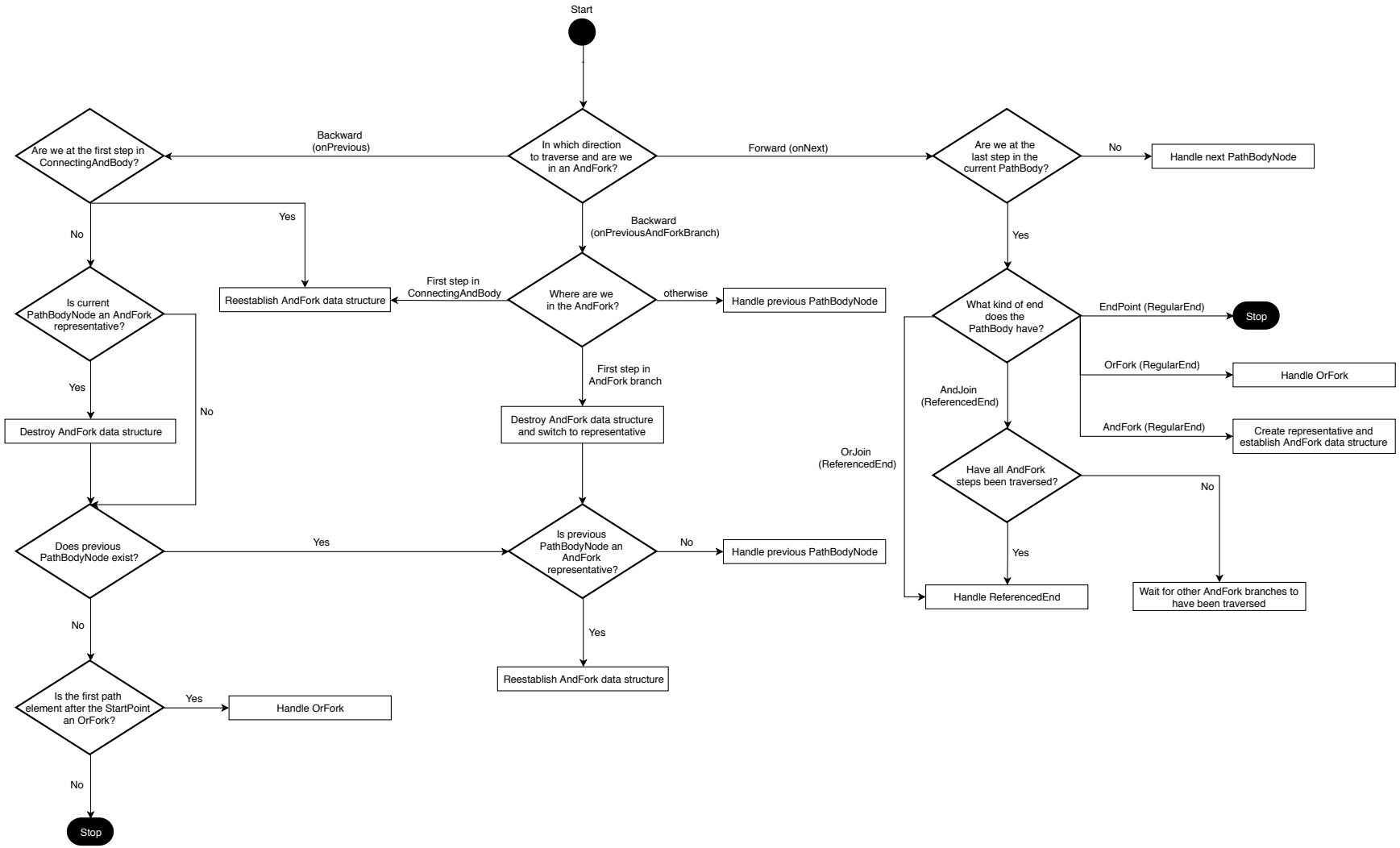


Figure 16: Flowchart describing traversal mechanism

4.3 Algorithm Overview

Before delving into the details of our algorithm in the following sections, we describe the overall structure of our algorithm using the flowchart in Fig. 16, capturing all cases that our traversal mechanism must be able to handle, corresponding to what a next or previous path element could be from a current one, and the more detailed pseudocode in Listing 3. As a reminder, a TUCM, which our algorithm steps through, begins with a StartPoint, which is followed by PathBodies containing zero or more PathBodyNodes, stored in our doubly linked list, or doubly linked lists for AndFork branches, as they are traversed, and ends with an EndPoint. The last path element in a PathBody following all PathBodyNodes in the PathBody can be either a RegularEnd, i.e., an AndFork, OrFork, or EndPoint, or a ReferencedEnd, i.e., an AndJoin or OrJoin. Hence, our algorithm starts by retrieving a TUCM's Path with its StartPoint, and first PathBody along with the PathBodyNodes it contains, if any. Then, at each PathBodyNode reached in the TUCM, we can choose to go to the previous traversed PathBodyNode or the PathBodyNode that follows the current one. Fig. 17 illustrates a UCM representing our forward and backward traversals described in this section.

Listing 3:
Pseudocode of traversal algorithm

```
if onNext
  if !current.isLastStepInCurrentPathBody()
    current = current.getNextPathBodyNodeInPathBody()
    current.retrieveStepInfo()
    current.getDoublyLinkedList().addLast(current)
  else
    regularEnd = current.getPathBody().getRegularEnd()
    if regularEnd != null
      handleRegularEnd(regularEnd)
    else
      if current.isWithinAndFork()
        if current.getAndFork().haveAllStepsBeenTraversed()
          connectingAndBody = current.getAndFork().getConnectingAndBody()
          handleConnectingBody(connectingAndBody)
      else if current.isWithinOrFork()
        connectingOrBody = current.getOrFork().getConnectingOrBody()
        handleConnectingBody(connectingOrBody)
else if onPrevious
  if current.isFirstStepInConnectingAndBody()
```

```
      current.getConnectingAndBody().getAndFork().reestablishAndFork()
      current.getDoublyLinkedList().removeLast()
  else
    if current.isAndForkRepresentative()
        current.removeAndForkLists()
    current = current.getDoublyLinkedList().getPreviousRemoveLast()
    if current == null
      firstPathBody = ucmMap.getPath().getPathBody()
      if firstPathBody.isEmpty() and firstPathBody.getRegularEnd().isOrFork()
        firstPathBody.getRegularEnd().getAlternativeSteps()
    else
      if current.isAndForkRepresentative()
        current.getAndFork().reestablishAndFork()
      else
        current.retrieveStepInfo()
else if onPreviousAndForkBranch
  if current.isWithinAndFork()
    if current.isFirstStepInConnectingAndBody()
      current.getConnectingAndBody().getAndFork().reestablishAndFork()
      current.getDoublyLinkedList().removeLast()
    else if current.isFirstStepInAndForkBranch()
      representative = current.getRepresentativePathBodyNode()
      current.removeAndForkLists()
      current = representative.getDoublyLinkedList().getPreviousRemoveLast()
      if current.isAndForkRepresentative()
        current.getAndFork().reestablishAndFork()
      else
        current.retrieveStepInfo()
    else
      current = current.getDoublyLinkedList().getPreviousRemoveLast()
      current.retrieveStepInfo()

handleConnectingBody(connectingBody):
if !connectingBody.isEmpty()
  current = connectingBody.getPathBodyNodes().get(0)
  current.retrieveStepInfo()
  current.getDoublyLinkedList().addLast(current)
else
  regularEnd = connectingBody.getRegularEnd()
  if regularEnd != null
    handleRegularEnd(regularEnd)
  else
    connectingBody = connectingBody.getParentConnectingBody()
    handleConnectingBody(connectingBody)

handleRegularEnd(regularEnd):
  switch (regularEnd)
  case: EndPoint
    exit
  case: OrFork
    regularEnd.getAlternativeSteps()
  case: AndFork
    regularEnd.getConcurrentSteps()
```

When traversing a TUCM forward (lines 1-17), where boolean variable `onNext` is true, several scenarios may occur. For more detailed examples, see Sections 4.5.1 and 4.5.3. First, if the current PathBodyNode is not the last one in the PathBody at hand, then we get the PathBodyNode that follows it within the PathBody (lines 2-5).

Otherwise, we know that we have reached the end of the PathBody (lines 6-17) and thus need to obtain the PathBody that follows it. The current PathBody has either a RegularEnd (i.e., an AndFork, OrFork, or EndPoint), or a ReferencedEnd (i.e., an AndJoin or OrJoin). Note that a null RegularEnd value signifies that we are at a ReferencedEnd.

If we are at a RegularEnd (lines 8-9), we use the function `handleRegularEnd` (lines 65-72) to either end the algorithm if the RegularEnd is an EndPoint, get the branch conditions of the alternative paths from which to select if the RegularEnd is an OrFork, or get the first PathBodyNode in each branch if the RegularEnd is an AndFork. In the case of an AndFork, once the first PathBodyNode of each of its branches is obtained, the representative PathBodyNode, corresponding to the first PathBodyNode of the first branch

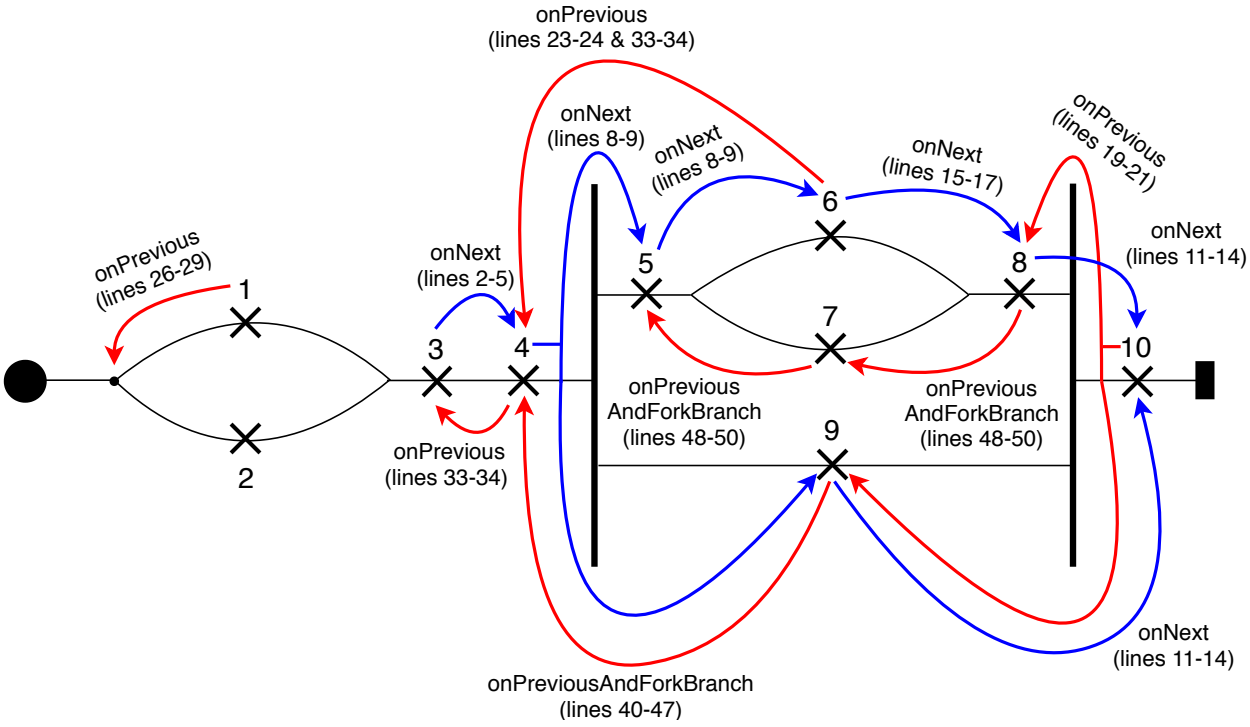


Figure 17: Example UCM demonstrating the forward (in blue) and backward (in red) traversals of TUCMs

in the AndFork which we step into, is added to the doubly linked list of the parent branch on which the AndFork lies as part of `getConcurrentSteps()`.

If we are at a ReferencedEnd (lines 10-17), we use the function `handleConnectingBody` (lines 52-63) to retrieve the first PathBodyNode, if it exists (lines 53-56), in the ConnectingAndBody or ConnectingOrBody, depending on whether we are in an AndFork (lines 11-14) or OrFork (lines 15-17), respectively. If we are in an AndFork, we must ensure that all the steps within it have been traversed prior to retrieving its ConnectingAndBody (line 12). In the event that a ConnectingBody does not contain any PathBodyNodes but has a RegularEnd (lines 59-60), we use the function `handleRegularEnd`, as explained earlier. Conversely, if the ConnectingBody is empty and does not have a RegularEnd (lines 61-63), then we recursively check the ConnectingBodies of the next outer AndFork or OrFork until we find a non-empty ConnectingBody or a ConnectingBody with a RegularEnd.

In addition, our algorithm encompasses two types of TUCM backward traversals, one for when we are on the TUCM's main Path but not inside any of its AndForks (lines 18-34), where boolean variable `onPrevious` is true, and the other for when we are within an AndFork (lines 35-50), where boolean variable `onPreviousAndForkBranch` is true. The motivation for having two backward navigation cases is to make it possible to traverse backward within an AndFork (e.g., from 7 to 5 in Fig. 17) and backward from anywhere within an AndFork to before the AndFork (e.g., from 6 to 4 in Fig. 17).

The `onPrevious` backward traversal checks whether we are at the first PathBodyNode in a ConnectingAndBody, and if we are (lines 19-21), it reestablishes the AndFork of the ConnectingAndBody, which signifies that we obtain the AndFork at the state it was left at before we moved to its ConnectingAndBody. In contrast, if we are not at the first PathBodyNode in a ConnectingAndBody, we first check whether we are at a PathBodyNode representing an AndFork, and if so (lines 23-24), we remove the doubly linked lists for the AndFork. We then get the previous PathBodyNode (i.e., the new current PathBodyNode) and remove the last PathBodyNode from the doubly linked list. If the new current PathBodyNode does not exist (lines 26-29), we check for a special case where an OrFork is the first path element in the TUCM after its StartPoint and retrieve the OrFork. If the new

current PathBodyNode exists (lines 30-34) and is the representative PathBodyNode of an AndFork (lines 31-32), then we reestablish the AndFork, otherwise we simply retrieve the information for the new last (current) PathBodyNode in the doubly linked list (lines 33-34). For more detailed examples, see Sections 4.5.2 and 4.5.4 except Fig. 36.

On the other hand, to iterate backward within an AndFork with the `onPrevious-AndForkBranch` traversal, we check if we are at the first PathBodyNode in a ConnectingAndBody and if so (lines 37-39), we reestablish the AndFork. If we are at the first step in an AndFork branch (lines 40-47), then we (a) remove the doubly linked lists for the AndFork, (b) get the previous PathBodyNode (i.e., the new current PathBodyNode) and remove the PathBodyNode representing the AndFork from the doubly linked list of the branch on which the AndFork lies, and, (c) if the new last (current) PathBodyNode of this parent branch is the representative PathBodyNode of an AndFork, we reestablish the AndFork, otherwise we obtain the information for the new last (current) PathBodyNode. Note that, as long as we are at the first PathBodyNode in one of the branches of an AndFork, it is possible to move back before the AndFork regardless of where we are in the other branches of the AndFork. Lastly, if we are neither at the first step in an AndFork branch or a ConnectingAndBody, then we simply get the information for the new (current) PathBodyNode in the PathBody (lines 48-50). For a more detailed example, see Fig. 36 in Section 4.5.4.

While it is necessary to recursively find the next ConnectingBody when traversing the TUCM in forward direction, we do not need to do so in backward direction, since the doubly linked lists always identify the previous path element.

4.4 Reading a TUCM Model

Our algorithm is contained within the Traversal class and begins by retrieving the XMI file of the TUCM models from a specified local path, and using Eclipse Modeling Framework (EMF) [36] classes to load the file as a resource from which the URNspec can be extracted. We then loop over the TUCMs contained within the URNspec until a TUCM

having a name matching one specified by the user by selecting a crisis situation is found. If found, we get the Path of the model, with its StartPoint and first PathBody, from which we derive PathBodyNodes. Afterwards, we follow the process detailed in the next section to get the first step on the Path.

4.5 Traversing a TUCM Model

We begin by describing the part of our algorithm dealing with the forward and then backward traversal of a TUCM's Path, excluding its AndForks, followed by the rest of our algorithm handling the forward and then backward traversal of a TUCM's Path, including its AndForks. We provide concrete examples of key scenarios that arise during these traversals in forward and backward direction.

When retrieving the information about a step, or PathBodyNode, we get the step type of the PathBodyNode and adapt the step's description according to the properties tied to its type, with a RegularStep simply storing the label of the step. For a ResourceStep, the step's description includes whether or not the ResourceType is locatable, using the keywords "Find" and "Get" respectively, followed by the name and the quantity needed of the ResourceType. As for a CommunicationStep, the step's description begins with the word "Contact" and is followed by either the name of the GroupType, if the ActorRoleType is a GroupType, or the name of the ActorRoleType itself, if it is not a GroupType. Other step information, i.e., its LocationType and Fact, if existent, is also obtained.

On the other hand, when reaching a RegularEnd of type EndPoint, we retrieve the latter's label. If the type of the reached RegularEnd is OrFork, we get the list of conditions of its branches, check which OrFork branch condition matches the one selected by the user, and get the PathBody corresponding to the branch with the chosen condition.

4.5.1 Forward TUCM Path Traversal without AndFork

For the forward traversal through a TUCM's Path excluding its AndForks, we present the list of key cases below regarding what the next path element of a current one could be

and then discuss each case individually.

1) retrieving the next PathBodyNode on a PathBody, if not all of its PathBodyNodes have been traversed yet (see Fig. 18)

2) retrieving the RegularEnd (EndPoint or OrFork) of a PathBody whose PathBodyNodes have all been traversed (see Figs. 19 and 20)

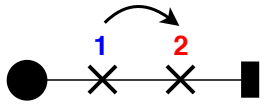
3) retrieving the ConnectingOrBody from the ReferencedEnd (OrJoin) of an OrFork (see Fig. 21)

4) retrieving the ConnectingOrBody from the ReferencedEnd (OrJoin) of an outer OrFork from a nested OrFork (see Fig. 22)

5) retrieving the RegularEnd (EndPoint or OrFork) of the ConnectingOrBody of an OrFork (see Figs. 23 and 24)

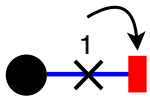
To iterate forward through the steps in the model's Path, we store the steps that have been completed in the doubly linked list. We compare the index number of the step reached within the current PathBody and the total number of PathBodyNodes within the current PathBody to determine whether we have reached the last step, or end, of the current PathBody. If we are not at the last PathBodyNode in a PathBody (case 1), we retrieve the next step, or PathBodyNode, in the PathBody (see Fig. 18), and add the new PathBodyNode to the doubly linked list of the TUCM's main Path.

On the other hand, if we have reached the last step, or end, of the current PathBody, we need to seek the next PathBody to step through, which can either be a RegularEnd (case 2), i.e., an EndPoint, OrFork, or AndFork (discussed later in the chapter), or a ReferencedEnd



Main Doubly Linked List:
Before: 1
After: 1 ⇄ 2

Figure 18: Example UCM for retrieving the following PathBodyNode (in red) from a current PathBodyNode (in blue) in a PathBody, including the doubly linked list of the UCM's main Path before and after retrieving the next PathBodyNode



Main Doubly Linked List:
Before: 1
After: 1

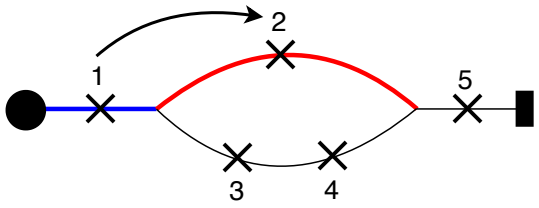
Figure 19: Example UCM for retrieving the RegularEnd of type EndPoint of a PathBody (in red) from the last PathBodyNode in the PathBody (in blue), including the doubly linked list of the UCM's main Path before and after retrieving the EndPoint

(case 3), i.e., an OrJoin or AndJoin (discussed later in the chapter). In case 3, we must seek the ConnectingOrBody or ConnectingAndBody, respectively.

For case 2, a PathBody having an EndPoint as RegularEnd (see Fig. 19) indicates the end of the TUCM, and thus the traversal algorithm would be done.

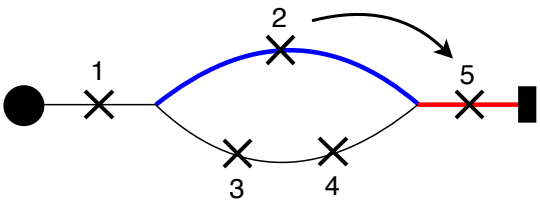
If a PathBody has an OrFork as RegularEnd (see Fig. 20), then we select the condition of the branch on which we want to proceed, and retrieve the first PathBodyNode on this branch and add it to the TUCM's main Path doubly linked list, if the OrFork branch is not empty. If it is empty, then we proceed to the ConnectingOrBody of the OrFork.

For case 3, we deal with situations where a PathBody has a ReferencedEnd instead of a RegularEnd. If a PathBody is part of an OrFork and hence has a ReferencedEnd of type OrJoin, we get the new PathBody, i.e., the ConnectingOrBody. If the ConnectingOrBody contains at least one PathBodyNode, we retrieve its first PathBodyNode and add the



Main Doubly Linked List:
Before: 1
After: 1 ⇄ 2

Figure 20: Example UCM for retrieving the first PathBodyNode, if any, in a selected branch of the RegularEnd of type OrFork of a PathBody (in red) from the last PathBodyNode in the PathBody (in blue), including the doubly linked list of the UCM's main Path before and after retrieving the next PathBodyNode, if any

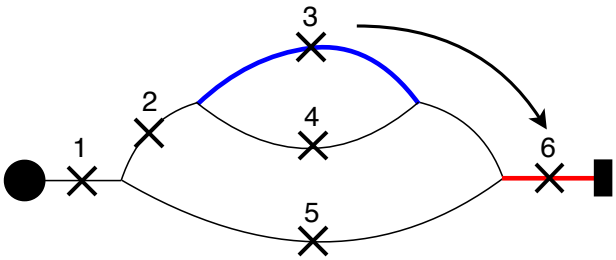


Main Doubly Linked List:
Before: 1 ⇄ 2
After: 1 ⇄ 2 ⇄ 5

Figure 21: Example UCM for retrieving the first PathBodyNode in a ConnectingOrBody (in red) from the last PathBodyNode of a branch in the ConnectingOrBody's OrFork (in blue), including the doubly linked list of the UCM's main Path before and after retrieving the next PathBodyNode

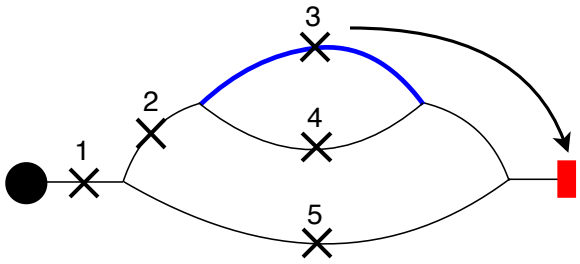
PathBodyNode to the doubly linked list of the TUCM's main Path (see Fig. 21).

Conversely, if the ConnectingOrBody does not contain any PathBodyNodes, we check whether it has a RegularEnd. If the empty ConnectingOrBody does not have a RegularEnd (case 4), i.e., it has a ReferencedEnd, we recursively get the ConnectingOrBody of all outer OrForks until we find a non-empty ConnectingOrBody (see Fig. 22), whose first PathBodyNode is added to the doubly linked list of the TUCM's main Path, or a ConnectingOrBody with a RegularEnd. Note that a ConnectingAndBody may be encountered during the recursion instead of a ConnectingOrBody, which is discussed in Section 4.5.3.



Main Doubly Linked List:
Before: 1 ⇄ 2 ⇄ 3
After: 1 ⇄ 2 ⇄ 3 ⇄ 6

Figure 22: Example UCM for retrieving the first PathBodyNode in the ConnectingOrBody of an OrFork (in red) from the last PathBodyNode of a branch in the OrFork's nested OrFork (in blue), including the doubly linked list of the UCM's main Path before and after retrieving the next PathBodyNode

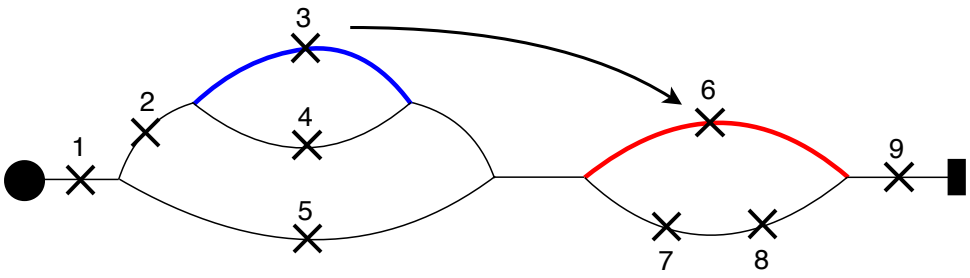


Main Doubly Linked List:
Before: 1 ⇄ 2 ⇄ 3
After: 1 ⇄ 2 ⇄ 3

Figure 23: Example UCM for retrieving the RegularEnd of type EndPoint of the ConnectingOrBody of an OrFork (in red) from the last PathBodyNode of a branch in the OrFork's nested OrFork (in blue), including the doubly linked list of the UCM's main Path before and after retrieving the EndPoint

If any empty ConnectingOrBody has a RegularEnd (case 5), we determine whether an EndPoint, OrFork, or AndFork (discussed later in the chapter) follows the ConnectingOrBody. For an EndPoint (see Fig. 23), the traversal mechanism ends.

In the case of an empty ConnectingOrBody ending with a RegularEnd of type OrFork (see Fig. 24), we choose the branch condition of the alternative path we want to continue our traversal on, and retrieve this branch's first PathBodyNode and add the latter to the TUCM's main Path doubly linked list, if the OrFork branch is not empty.



Main Doubly Linked List:
Before: 1 ⇄ 2 ⇄ 3
After: 1 ⇄ 2 ⇄ 3 ⇄ 6

Figure 24: Example UCM for retrieving the first PathBodyNode, if any, in a selected branch of the RegularEnd of type OrFork of the ConnectingOrBody of an OrFork (in red) from the last PathBodyNode of a branch in the OrFork's nested OrFork (in blue), including the doubly linked list of the UCM's main Path before and after retrieving the next PathBodyNode, if any

4.5.2 Backward TUCM Path Traversal without AndFork

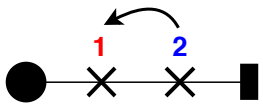
For the backward navigation through a TUCM's Path excluding its AndForks, we have the two key cases below for obtaining the previous path element from a current one, which we then describe.

1) stepping back from one PathBodyNode to the one that precedes it in a PathBody (see Fig. 25)

2) stepping back from the first PathBodyNode in the branch of an OrFork, representing the first path element after the StartPoint in a TUCM, to the OrFork (see Fig. 26)

To begin, we remove the last PathBodyNode in the doubly linked list of the TUCM's main Path. If the doubly linked list still contains PathBodyNodes after this deletion, we know that stepping back in the TUCM is possible. Hence, we retrieve the new last PathBodyNode in the doubly linked list (case 1), as shown in Fig. 25.

In contrast, if the doubly linked list becomes empty, we have to check for a special case. Although we originally had one PathBodyNode in the doubly linked list, which would normally signify that we should not be able to go backward in the TUCM as no previous step exists, we are able to navigate backward in this special case where the first path element in the TUCM after its StartPoint is an OrFork (case 2). In this case, we must retrieve the OrFork again to be able to choose another alternative path to traverse (see Fig. 26). Once the user has decided, the first PathBodyNode from the corresponding branch of the OrFork is added to the doubly linked list as its first element.



Main Doubly Linked List:
Before: 1 ⇄ 2
After: 1

Figure 25: Example UCM for stepping back from one PathBodyNode (in blue) to the PathBodyNode that precedes it (in red) in a PathBody, including the doubly linked list of the UCM's main Path before and after retrieving the previous PathBodyNode

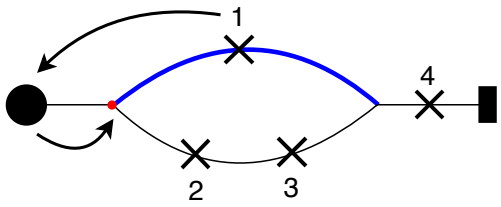


Figure 26: Example UCM for stepping back from the first PathBodyNode in the branch of an OrFork (in blue), representing the first path element in the UCM after its StartPoint, to the OrFork (in red) to choose an alternative path, including the doubly linked list of the UCM's main Path before and after retrieving the OrFork

4.5.3 Forward TUCM Path Traversal with AndFork

For the forward traversal through a TUCM's AndForks, we provide the list of key cases below related to crossing into or out of an AndFork on a TUCM's main Path and then discuss them individually.

1) retrieving the RegularEnd (AndFork) of a PathBody whose PathBodyNodes have all been traversed (see Fig. 27)

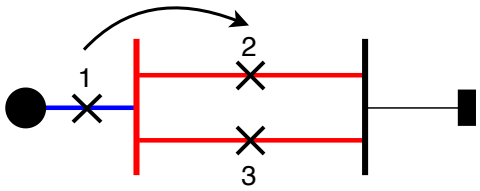
2) retrieving the RegularEnd (AndFork) of the ConnectingOrBody of an OrFork (see Fig. 28)

3) retrieving the ConnectingAndBody from the ReferencedEnd (AndJoin) of an AndFork (see Fig. 29)

4) retrieving the RegularEnd (EndPoint, OrFork, or AndFork) of the ConnectingAndBody of an AndFork (see Figs. 30, 31, and 32)

5) recursing through a ConnectingAndBody and a ConnectingOrBody (see Fig. 33)

When crossing into an AndFork in forward direction (i.e., if the RegularEnd of a PathBody is of type AndFork), we retrieve the first PathBodyNode on each of its branches, which will always work given our assumption of adding empty PathBodyNodes at the beginning of AndFork branches that begin with an OrFork or AndFork instead of a



Doubly Linked Lists:

Before:
Main list: 1

After:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

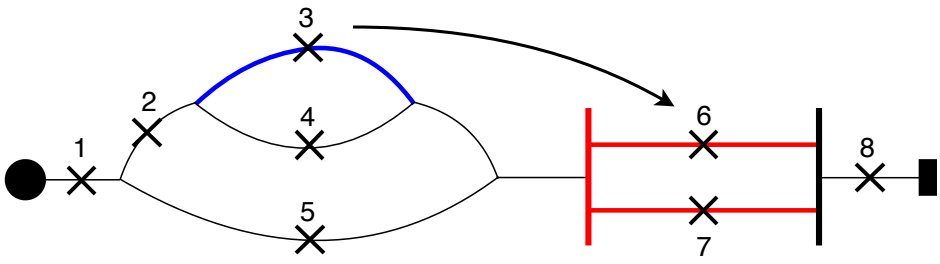
Figure 27: Example UCM for retrieving the first PathBodyNode of each branch in the RegularEnd of type AndFork of a PathBody (in red) from the last PathBodyNode in the PathBody (in blue), including the doubly linked lists of the UCM's main Path and AndFork branches before and after retrieving the AndFork

PathBodyNode.

When crossing out of an AndFork (i.e., past its corresponding AndJoin), we check that all branches of the AndFork have arrived at the AndJoin. If this is not the case, forward traversal is not allowed.

To iterate forward in a TUCM containing AndForks, when the PathBodyNodes of a PathBody have all been traversed and the PathBody has a RegularEnd of type AndFork (case 1), we retrieve the first PathBodyNode in each of the branches of the AndFork (see Fig. 27). Then, the first PathBodyNode of the first branch in the AndFork is added to the doubly linked list of the TUCM's main Path such that we are able to tell when an AndFork has been traversed in case we want to step back into it while navigating backward through the model. We also create a doubly linked list for each branch and add the first PathBodyNode on each branch to its respective doubly linked list.

Similarly, if we are at the end of a branch of an OrFork whose ConnectingOrBody has a RegularEnd of type AndFork (case 2), we also retrieve the first PathBodyNode in each of the branches of the AndFork, add that of the first branch to the doubly linked list of the TUCM's main Path, create a doubly linked list for each branch, and add the first PathBodyNode on each branch to its respective doubly linked list (see Fig. 28).



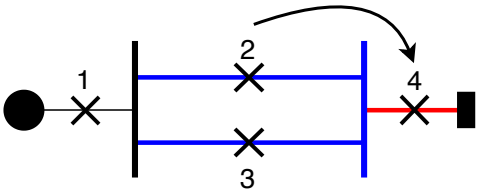
Doubly Linked Lists:

Before:
Main list: 1 ⇄ 2 ⇄ 3

After:
Main list: 1 ⇄ 2 ⇄ 3 ⇄ 6
Top AndFork branch (connected to 6 in main list): 6
Bottom AndFork branch (connected to 6 in main list): 7

Figure 28: Example UCM for retrieving the first PathBodyNode of each branch in the RegularEnd of type AndFork of the ConnectingOrBody of an OrFork (in red) from the last PathBodyNode of a branch in the OrFork's nested OrFork (in blue), including the doubly linked lists of the UCM's main Path and AndFork branches before and after retrieving the AndFork

Then, when leaving an AndFork on the TUCM's main Path, once all of its steps have been traversed, we get the new PathBody, i.e., the ConnectingAndBody, from the ReferencedEnd of type AndJoin (case 3). If the ConnectingAndBody contains at least one



Doubly Linked Lists:

Before:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

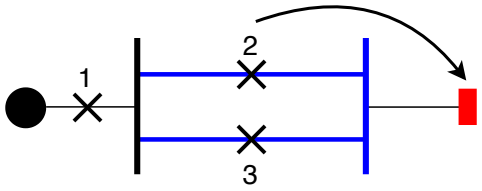
After:
Main list: 1 ⇄ 2 ⇄ 4
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

Figure 29: Example UCM for retrieving the first PathBodyNode in a ConnectingAndBody (in red) from its AndFork when all the AndFork's steps have been traversed (in blue), including the doubly linked lists of the UCM's main Path and AndFork branches before and after retrieving the next PathBodyNode

PathBodyNode, we retrieve its first PathBodyNode and add it to the doubly linked list of the TUCM's main Path (see Fig. 29).

On the other hand, if the ConnectingAndBody is empty and has a RegularEnd (case 4), we get the path element following the empty ConnectingAndBody based on the type of the RegularEnd. If the RegularEnd is of type EndPoint (see Fig. 30), the traversal algorithm comes to an end. If the RegularEnd is of type OrFork (see Fig. 31), we select the condition of the branch of the alternative path we wish to traverse, and retrieve this branch's first PathBodyNode and add the latter to the doubly linked list of the TUCM's main Path, if the OrFork branch is not empty. Otherwise, if the RegularEnd is of type AndFork (see Fig. 32), we retrieve the first PathBodyNode in each of the branches of the AndFork, add the PathBodyNode of the first branch to the doubly linked list of the TUCM's main Path, create a doubly linked list for each branch, and add the first PathBodyNode on each branch to its respective doubly linked list.

If the ConnectingAndBody is empty and does not have a RegularEnd, then the next outer ConnectingBody needs to be found recursively (case 5). This is discussed for OrForks

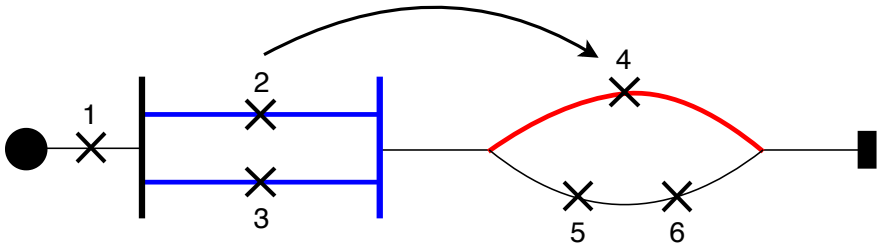


Doubly Linked Lists:

Before:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

After:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

Figure 30: Example UCM for retrieving the RegularEnd of type EndPoint of a ConnectingAndBody (in red) from its AndFork when all the AndFork's steps have been traversed (in blue), including the doubly linked lists of the UCM's main Path and AndFork branches before and after retrieving the EndPoint

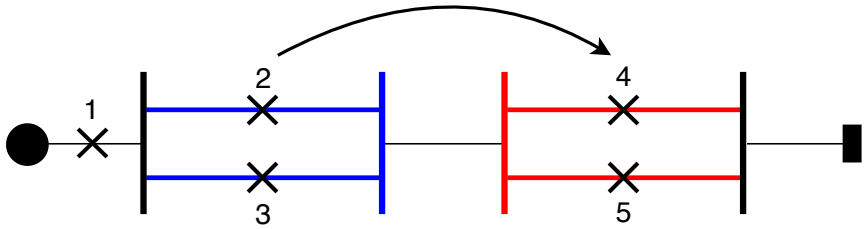


Doubly Linked Lists:

Before:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

After:
Main list: 1 ⇄ 2 ⇄ 4
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

Figure 31: Example UCM for retrieving the first PathBodyNode, if any, in a selected branch of the RegularEnd of type OrFork of a ConnectingAndBody (in red) from its AndFork when all the AndFork's steps have been traversed (in blue), including the doubly linked list of the UCM's main Path and AndFork branches before and after retrieving the next PathBodyNode, if any



Doubly Linked Lists:

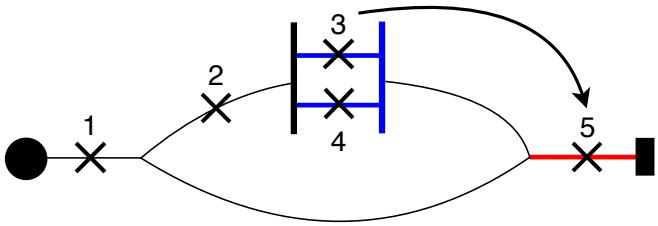
Before:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

After:
Main list: 1 ⇄ 2 ⇄ 4
First AndFork top branch (connected to 2 in main list): 2
First AndFork bottom branch (connected to 2 in main list): 3
Second AndFork top branch (connected to 4 in main list): 4
Second AndFork bottom branch (connected to 4 in main list): 5

Figure 32: Example UCM for retrieving the first PathBodyNode of each branch in the RegularEnd of type AndFork of a ConnectingAndBody (in red) from its AndFork when all the AndFork's steps have been traversed (in blue), including the doubly linked lists of the UCM's main Path and AndFork branches before and after retrieving the AndFork

in Section 4.5.1, and equally applies to AndForks, except that we need to check that all branches have arrived at an AndJoin to proceed beyond the AndJoin. Furthermore, it is possible that ConnectingOrBodies and ConnectingAndBodies are encountered during the recursion as shown in Fig. 33. In Fig. 33, the first PathBodyNode of the next outer non-empty ConnectingBody (a ConnectingOrBody in this case) is retrieved and added to the doubly linked list of the TUCM's main Path.

When navigating forward through AndFork branches (i.e., not on a TUCM's main Path), most cases in Section 4.5.1 for the TUCM Path traversal without AndForks, as well as the cases listed above, apply. However, the cases where the RegularEnd of a current PathBody is of type EndPoint are not applicable in the traversal of AndFork branches, since the traversed TUCMs are always well-nested. Another difference is that traversing an AndFork branch in forward direction adds the next PathBodyNode reached to the doubly linked list of the branch, as opposed to the doubly linked list of the TUCM's main Path.



Doubly Linked Lists:

Before:
Main list: 1 ⇄ 2 ⇄ 3
Top AndFork branch (connected to 3 in main list): 3
Bottom AndFork branch (connected to 3 in main list): 4

After:
Main list: 1 ⇄ 2 ⇄ 3 ⇄ 5
Top AndFork branch (connected to 3 in main list): 3
Bottom AndFork branch (connected to 3 in main list): 4

Figure 33: Example UCM for retrieving the first PathBodyNode in a ConnectingOrBody (in red) from an AndFork, when all the AndFork's steps have been traversed, in a branch of the ConnectingOrBody's OrFork (in blue), including the doubly linked lists of the UCM's main Path and AndFork branches before and after retrieving the next PathBodyNode

4.5.4 Backward TUCM Path Traversal with AndFork

For the backward navigation through a TUCM's Path with its AndForks, we have two distinct types of backward traversals. The first steps back on the TUCM's main Path where either the current PathBodyNode or the previous PathBodyNode is not inside any of its AndForks. The second is used when we remain within an AndFork (i.e., the current PathBodyNode and previous PathBodyNode are both within an AndFork). We implement these two backward navigations to allow us to traverse backward from anywhere within an AndFork to before the AndFork on the TUCM's main Path and within an AndFork, respectively. We present key cases below related to obtaining the previous path element from a current one, followed by a description of each case.

For the backward traversal to and from an AndFork on a TUCM's Path:

1) stepping back from a ConnectingAndBody across an AndJoin into an AndFork (see Fig. 34)

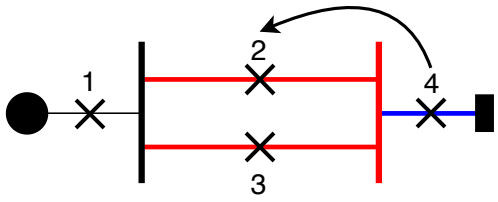
2) stepping from anywhere within an AndFork back to before the AndFork on the TUCM's main Path (see Fig. 35)

For the backward traversal within an AndFork:

3) stepping back from a nested AndFork to the parent AndFork branch on which the nested AndFork lies (see Fig. 36)

In case 1, we step back from the first PathBodyNode of the ConnectingAndBody of an AndFork on a TUCM's main Path back into the AndFork by reestablishing the AndFork, which means that we get the state it was left at (i.e., the last step for each branch of the AndFork) before we obtained its ConnectingAndBody, and then we remove the last PathBodyNode, i.e., the first PathBodyNode in the ConnectingAndBody, from the doubly linked list of the TUCM's main Path (see Fig. 34).

In case 2, to step back from anywhere within an AndFork to before the AndFork on the TUCM's main Path, thereby eliminating our progress through the AndFork, we remove the last PathBodyNode from the doubly linked list of the TUCM's main Path, i.e., we



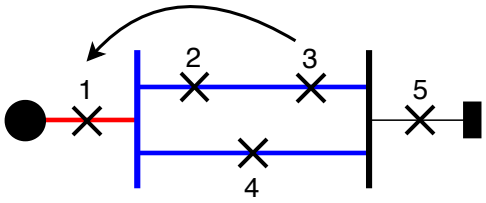
Doubly Linked Lists:

Before:
Main list: 1 ⇄ 2 ⇄ 4
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

After:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2
Bottom AndFork branch (connected to 2 in main list): 3

Figure 34: Example UCM for stepping back from the first PathBodyNode in a ConnectingAndBody (in blue) to its AndFork (in red), including the doubly linked lists of the UCM's main Path and AndFork branches before and after reestablishing the AndFork

remove the representative PathBodyNode of the AndFork, and then retrieve the new last PathBodyNode in the doubly linked list (see Fig. 35). We also destroy the doubly linked lists for the AndFork.



Doubly Linked Lists:

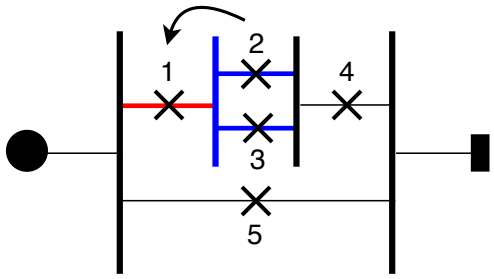
Before:
Main list: 1 ⇄ 2
Top AndFork branch (connected to 2 in main list): 2 ⇄ 3
Bottom AndFork branch (connected to 2 in main list): 4

After:
Main list: 1

Figure 35: Example UCM for stepping back from anywhere within an AndFork (in blue) to the last PathBodyNode on the TUCM's main Path before the AndFork (in red), including the doubly linked lists of the UCM's main Path and AndFork branches before and after retrieving the previous PathBodyNode

As for the backward traversal within an AndFork branch, the cases where we want to step back from (a) one PathBodyNode to the one that precedes it on the branch or (b) a ConnectingAndBody to its nested AndFork follow both the corresponding cases described for the backward traversal of a TUCM's Path with and without AndForks. However, in the latter case, we delete the last PathBodyNode, representing the first PathBodyNode in the ConnectingAndBody, from the doubly linked list of the branch on which the AndFork lies.

Case 3 describes the special case where we step back from the first step in a nested AndFork's branch to the parent AndFork branch on which the nested AndFork lies, as demonstrated in Fig. 36. In this case, we must (a) get the representative PathBodyNode of the nested AndFork, (b) remove it, i.e., the last PathBodyNode, from the doubly linked list of the parent branch, and (c) remove the doubly linked lists involved in the nested AndFork. This case also applies to moving back from the first step in a branch of the top-level AndFork in the TUCM's main Path to the step before the AndFork.



Doubly Linked Lists:

Before:
Main list: 1
(1) Top AndFork branch (connected to 1 in main list): 1 ⇄ 2
(2) Bottom AndFork branch (connected to 1 in main list): 5
(3) Top nested AndFork branch (connected to 2 in list 1): 2
(4) Bottom nested AndFork branch (connected to 2 in list 1): 3

After:
Main list: 1
Top AndFork branch (connected to 1 in main list): 1
Bottom AndFork branch (connected to 1 in main list): 5

Figure 36: Example UCM for stepping back from the first PathBodyNode in a nested AndFork's branch (in blue) to the last PathBodyNode in the parent AndFork branch (in red), including the doubly linked lists of the AndFork branches before and after retrieving the previous PathBodyNode

4.6 Summary

Our traversal algorithm, which allows the forward and backward navigation of crisis management TUCMs authored by crisis experts, is detailed in this chapter. The following chapter uses our crisis management DSL and our traversal algorithm as basis for our proof-of-concept implementation, an Android mobile application, which aims to validate the feasibility of our DSL and the correctness of our traversal mechanism, by allowing the visualization of each step in a model on the user interface, and moving to the next or previous step.

Chapter 5

Application of the DSL for Crisis Management

In this chapter, as our proof-of-concept implementation, we discuss a system designed based on the textual implementation of our crisis management domain-specific software language (DSL) and our traversal algorithm, and used by crisis experts, victims, and first responders.

5.1 Technical Overview

An overview of the proposed underlying software architecture is provided in Fig. 37 which illustrates the role of a user, e.g., crisis expert, in creating models, i.e., extended Textual Use Case Maps (TUCMs) [1], employing our DSL Xtext editor, which are then accessed using our traversal algorithm and loaded into the mobile application we build to allow users to visualize the step-by-step navigation of the models in forward and backward

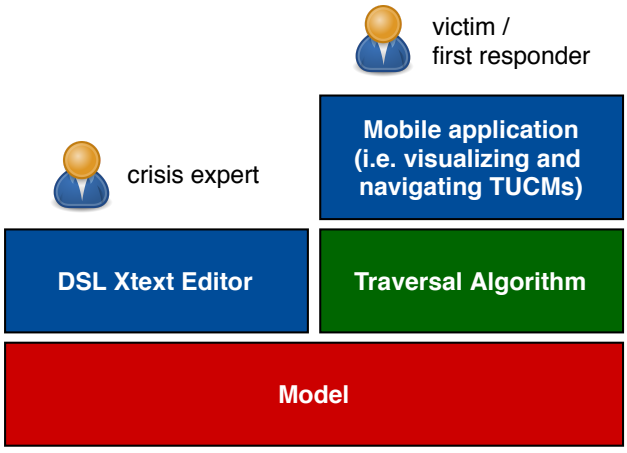


Figure 37: Overview of the architecture of the proposed software stack

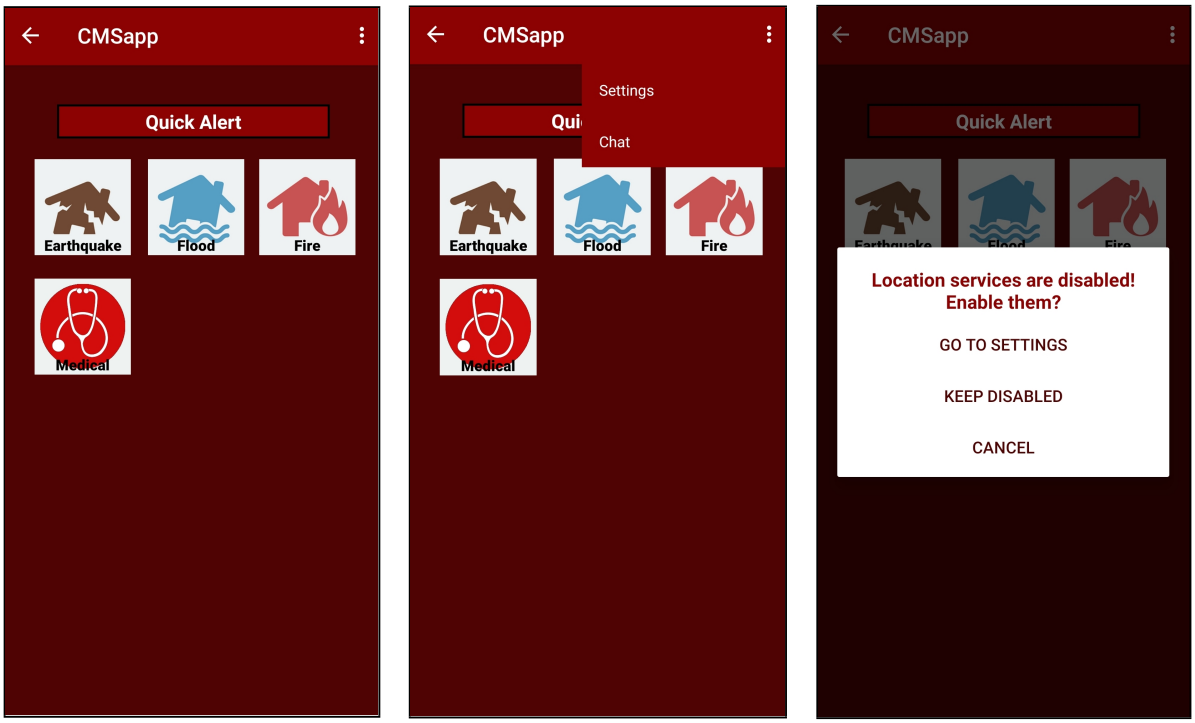
direction. In more detail, once the DSL for crisis management is created, crisis experts may use it to produce TUCMs illustrating emergency situation procedures, which can then be followed by authorities and victims during a crisis. Observing the TUCMs, crisis respondents and victims (R1) may know what actions to take and why (R5), what resources are needed, how they can be obtained (R2) and where they can be found (R6), what authorities or victims must be contacted (R3), and what actions they should be cautious about (R4).

Crisis experts define emergency procedures offline, using the classes in the type view of the metamodel and the extended syntax of the TUCM notation. Then, the TUCMs created by the crisis experts are converted into XML Metadata Interchange (XMI) [35] form. When an emergency actually occurs, we envision a mobile application that uses our algorithm to traverse the XMI file and display the set of steps to deal with the crisis at hand in order to help victims and crisis respondents through the emergency. The mobile application is built on the Android platform due to our familiarity with developing mobile applications on this technology.

The mobile application user first identifies the emergency which then triggers the mobile application to follow the corresponding emergency procedure, i.e., the classes in the execution view of the metamodel are instantiated and connected to the corresponding instances from the type view. Each step is presented sequentially to the user by running the traversal algorithm on the XMI file of the TUCM created by crisis experts and matching the emergency selected by the user. As the emergency unfolds, the actor, resource, and location types specified by the crisis expert are mapped to actual actors, resources, and locations. As much as possible, this information is determined by context without user involvement. In addition, the mobile application allows individuals to directly alert relevant authorities during an emergency situation, in order for the alerted authorities to determine what kind of help, e.g., authorities and resources, must be sent to manage the crisis situation. Furthermore, the mobile application allows users to connect with others that also have the mobile application, hence establishing ad-hoc networks for victims and authorities to communicate with each other more effectively.

5.2 Specific Features

The mobile application, named CMSapp where CMS refers to Crisis Management Systems, is used during emergencies to guide users on the actions to take. When a user starts the mobile application, they must select the type of emergency they are currently in, as shown in Fig. 38a, which will prompt them to allow SMS and Location permissions, depicted in Fig. 38c, if not already granted. For the Location permission, the user must enable this permission via the settings page in Fig. 39, accessed via the menu items as portrayed in Fig. 38b, by toggling the location services option, which will trigger the mobile application to check whether the permission has been allowed on the device for the mobile application, and if not, will request the permission from the user. Note that a user may choose to keep the permission to track their location disabled and still proceed to the display of the relevant emergency procedure. When the SMS and Location permissions



(a) (b) (c)

Figure 38: Application's (a) start-up page where users choose the type of emergency encountered, (b) view the available menu items, and (c) trigger a prompt to be displayed if the Location permission is disabled upon the selection of the emergency type

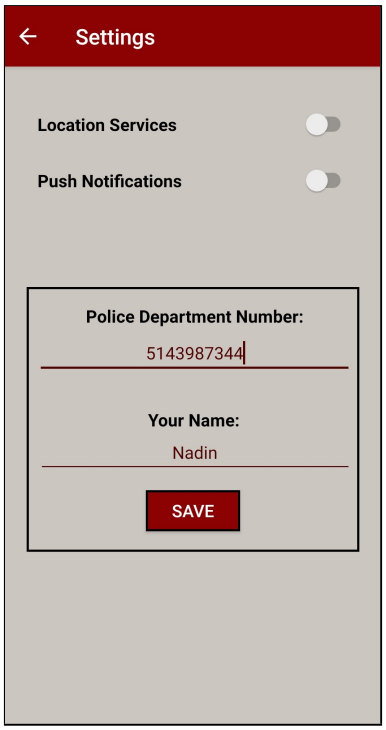
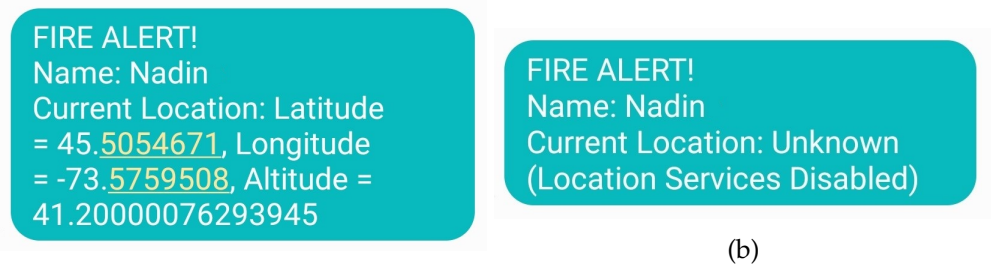


Figure 39: Settings Page

have been dealt with, the user's device built-in text messaging application automatically sends the user's name, as specified by the user in the settings page in Fig. 39, and location, as latitude, longitude, and altitude coordinates, if the permission was granted, to the phone number of the authority given as input by the user in the settings page, so that the relevant authority may be alerted of the emergency situation at hand. Figs. 40a and 40b present the content of the text message sent to alert the authority of the crisis depending on whether the Location permission is or is not granted by the user, respectively.



(a) (b)

Figure 40: SMS alert sent, with Location permission (a) enabled or (b) disabled, upon choosing the type of encountered emergency, in this case Fire

Figure 41: Emergency procedure displayed based on the type of the steps it encompasses, i.e., (a) regular Step, (b) ResourceStep for a resource to find, (c) ResourceStep for a resource to request, (d) WarningStep, and (e) CommunicationStep, with the different labels possible for the "Next" button, i.e., (a) "Location Update", (b) "Resource Found", (c) "Request Resource", (d) "Next", and (e) "Finish" once the user has reached the end of the emergency plan

Afterwards, the emergency procedure is finally revealed to the user step-by-step. A distinct icon helps distinguish the type of the current step being viewed, i.e., ResourceStep, CommunicationStep, or WarningStep, with no icon representing a regular Step. Figs. 41a, 41b, 41c, 41d, and 41e show a regular Step, ResourceStep for a resource to find, ResourceStep for a resource to request from the authority, WarningStep, and CommunicationStep, respectively. When the CommunicationStep icon is clicked, the user's device built-in calling application is automatically opened with the phone number of the authority specified by the user in the settings page already dialed, as depicted in Fig. 42, in order for the user to contact the authority. Furthermore, if applicable, the concept of a "fact" is shown in grey, in italic, and within parentheses, below the description of a step, as in Fig. 41d. As the user progresses through the steps in the emergency plan by pressing the "Next" button, the latter's text may change, as demonstrated in Fig. 41, to denote steps where important information is to be sent to the authority via text message upon the click of the button. The contents of these text messages are based on the label of the button when it is clicked and

Figure 42: Device built-in calling application opens, with the phone number of the authority from the settings page already dialed, when the icon in a CommunicationStep is clicked

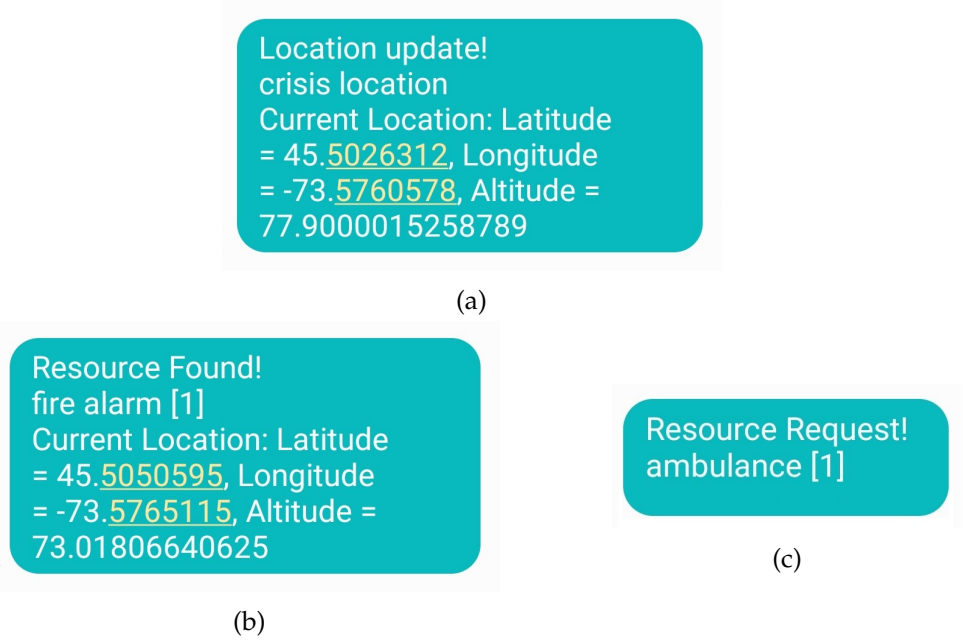
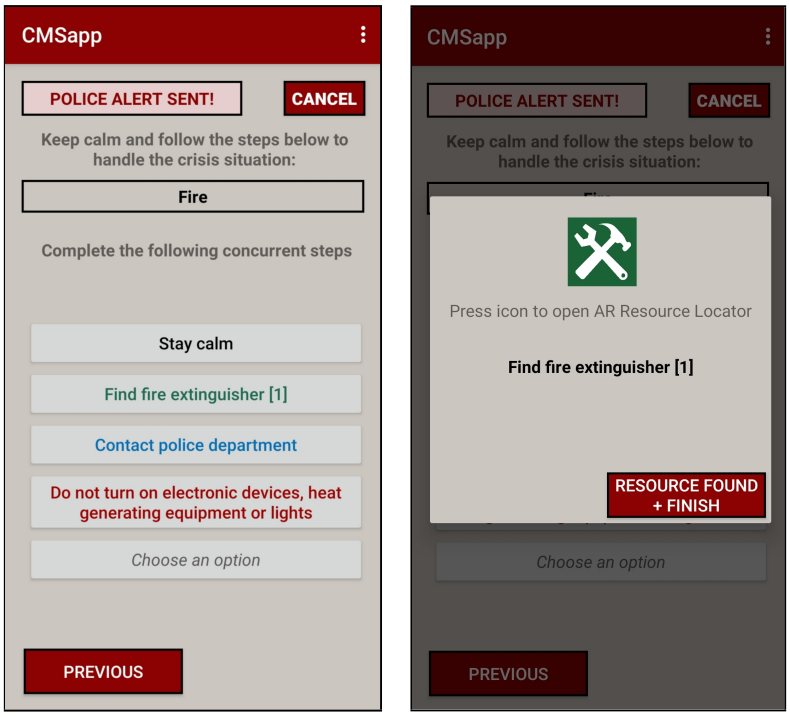


Figure 43: SMS sent depending on whether the "Next" button's label is (a) "Location Update", (b) "Resource Found", or (c) "Request Resource"

are shown in Fig. 43. More specifically, if the button states "Location Update" (see Fig. 41a), the authority will receive the name and coordinates, if the Location permission is enabled, of the current location of the user which would have differed from their last sent location (see Fig. 43a). On the other hand, "Resource Found" (see Fig. 41b) will send the user's current location, if the permission has been granted, as well as the name and quantity of the resource which was found by the user (see Fig. 43b), and "Request Resource" (see Fig. 41c) will request a resource, not locatable by the user, from the authority while providing the quantity of the resource needed to deal with the emergency situation at hand (see Fig. 43c).

In addition to iterating forward through the emergency plan's steps, the user may also iterate backward using the "Previous" button, as shown in Fig. 41. Any individual step is displayed on the interface alone. However, when depicting concurrency, as portrayed in Fig. 44a, each step happening in parallel is represented on the interface by a button having the step's description as label. A label in black, green, blue, or red represents a regular Step, a ResourceStep, a CommunicationStep, or a WarningStep, respectively. On the other hand, an OrFork is denoted by a button with the "Choose an option" label in light grey and italic font. When a parallel button is clicked, a dialog box opens (see Fig. 44b)



(a) (b)

Figure 44: (a) An AndFork is represented as a button per concurrent step, where a label in black, green, blue, red, or grey and italic represents a regular Step, ResourceStep, CommunicationStep, WarningStep, or OrFork, respectively. (b) A dialog is displayed with the step's details when the step's button is clicked

that provides the step's details, similar to the information displayed when a single step is viewed, and from which the user can choose to proceed to the previous or next step, if they exist, in that parallel path. For OrForks, the dialog box prompts the user to select, from a dropdown menu, the condition of the branch in the OrFork corresponding to the path they want to follow, as in Fig. 45. Once all concurrent steps are complete, the "Next" button becomes visible to allow the user to continue the steps in the main path of the emergency plan. For the cases where alternative paths are present, upon clicking on the "Next" button, a dialog box is displayed, identical to that shown in Fig. 45, allowing the selection of the OrFork branch condition specific to the path the user wishes to follow to continue the emergency plan. Once the end of the plan is reached, the "Next" button is replaced by a "Finish" button (see Fig. 41e) which the user may click to inform the authority that the crisis has been handled on the user's end, as illustrated in Figs. 46a and 46b, and go back to the mobile application's screen at start-up. Moreover, at any given time while

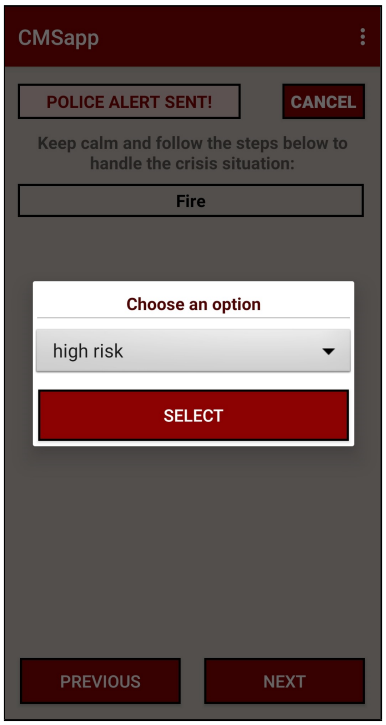
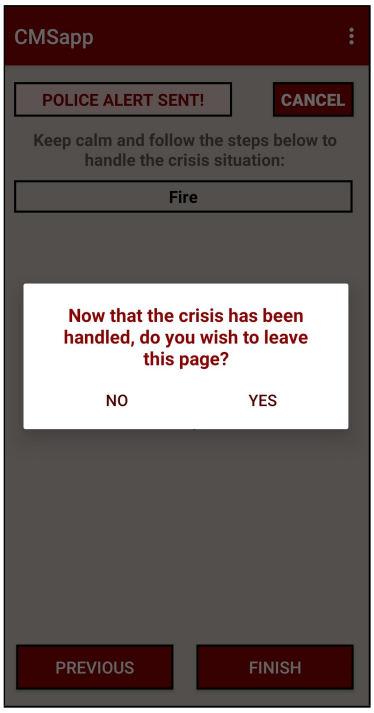


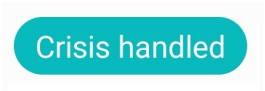
Figure 45: At an OrFork, one alternative path is followed by selecting its branch condition from a dialog's dropdown menu

the emergency procedure is open, a crisis alert may be cancelled via the "Cancel" button (see Fig. 41), which automatically sends a text message to the authority informing them of the crisis' cancellation, as can be seen in Figs. 46c and 46d.

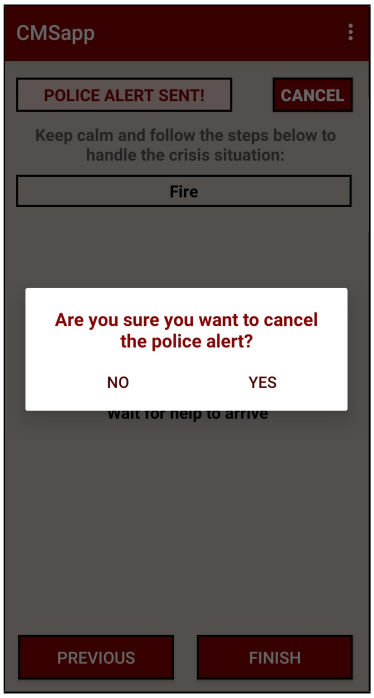
Besides the settings page in the mobile application's menu items, a page for an instant messaging chat among the users of the mobile application is available (see Fig. 38b). For the chat to open, if never accessed before, the user must provide a valid phone number in order to receive a verification code via text message, which the user will need to enter in the mobile application to validate his or her chat login, as depicted in Fig. 47a. Once the user has been authorized and added to the mobile application's chat user database, the user will see a list of the different types of emergencies handled by the mobile application as in Fig. 47b, and by clicking any of them, they may open the chat corresponding to the selected emergency type and communicate with other mobile application users, or simply read messages exchanged in the past by other users. Fig. 48 shows an example of two mobile application users messaging each other via the Fire chat on their respective devices.



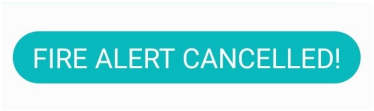
(a)



(b)

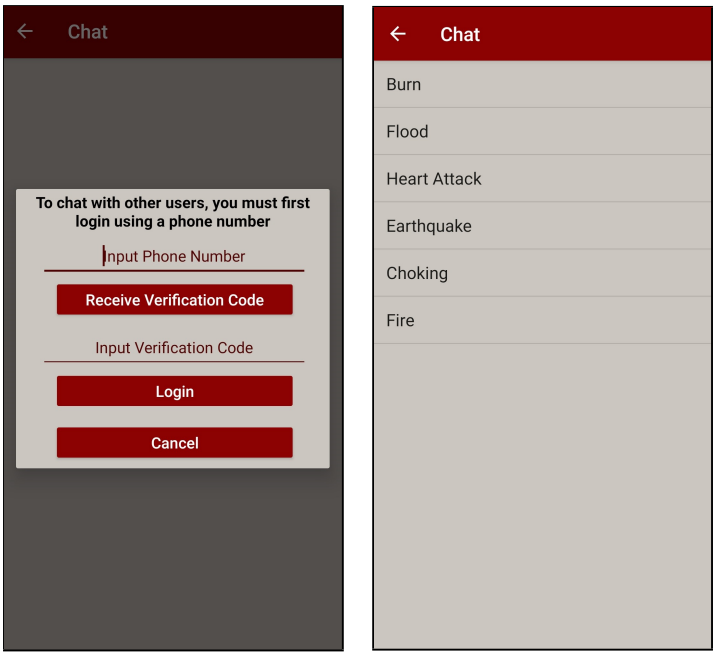


(c)



(d)

Figure 46: (a) A dialog appears when the "Finish" button is clicked to check whether the crisis has really been handled, and if confirmed, (b) a text message is sent to inform the authority that the user has completed the emergency procedure and that the crisis has been handled on their end. (c) A dialog appears when the "Cancel" button is clicked to check whether the user is sure that they want to cancel the crisis alert, and if they do, (d) a text message is sent to inform the authority that the crisis alert was cancelled



(a) (b)

Figure 47: Chat among mobile application users requires (a) logging in by providing a valid phone number for a device's first time access, and once authorized, (b) the emergency types handled by the mobile application are listed and the chat corresponding to an emergency type opens upon selection

(a) (b)

Figure 48: Example of two mobile application users, (a) Nadin and (b) Anna, chatting with each other via the Fire chat on their respective devices

5.3 Future Improvements

In the future, to more effectively use resources, the envisioned mobile application could use augmented reality features to guide users toward the nearest available resource specified in the extended TUCM, provided that the resource is locatable, i.e., it can be found near the location of a disaster, thereby taking the 3D environment of a crisis into account. This feature can be achieved by adding functionality to the icon already included in the user interface of a ResourceStep (see Fig. 41b), similar to the one of a CommunicationStep. Another feature could enable the mobile application to be used outside emergencies, in addition to during emergencies, by allowing users to explore emergency procedures any time to be better prepared for them in the future. This component of the mobile application is not currently supported since a text message is automatically sent to a specified authority to alert them of an encountered crisis as soon as the user selects an emergency to visualize the steps to manage it, which would not be appropriate outside a crisis since a text message would be sent without the existence of any actual threat. Furthermore, although the current system includes a toggle for enabling or disabling push notifications in the settings page (see Fig. 39), this feature is not fully implemented, but would be beneficial to inform users of an emergency nearby if an alert has already been sent by a mobile application user close to them, or whenever new messages are received in the chat. Another feature which would improve the mobile application would be to add the possibility of storing multiple contact phone numbers in the settings page so that the specified authorities or victims may be contacted via call or text message in parts of the emergency procedure in which they are relevant. At the present time, only one contact number, that of the police department, can be saved by the user in the settings page (see Fig. 39), and thus any step which includes calling or sending a text message does so by dialing or texting the one stored phone number. However, for example, if it would be preferable to contact the fire department directly during a certain crisis incident but only the police department's number is stored in the mobile application, then the latter would be contacted, which may cause help from the fire department to take longer to arrive. In addition, the mobile application assumes

that the workflow displayed for a type of emergency deals with one specific role type. Nevertheless, if the mobile application could be extended to handle more role types by allowing users to select their role type when identifying the emergency situation at the start of the mobile application, the emergency procedure targeting the specific role the user is playing in a given crisis event could then be displayed. Past work proposes a promising DSL for role-based access control applied in disaster relief intervention, where resources are either granted or denied to users based on their roles, thereby ensuring security and quality of service [37], which represent requirements beyond the scope of the current work, as we focus solely on the core crisis management concepts underlying our proposed DSL, but would be valuable if given future consideration. Another minor aspect of the mobile application that can be enhanced involves the feature of cancelling a crisis alert. Having a prompt asking the user to provide the reason for the cancellation prior to allowing the crisis to be cancelled would inform the authority of the rationale behind the cancellation, e.g., whether the crisis alert was accidentally sent, or whether an incident was mistaken as being an emergency when in fact it is not. Additionally, the mobile application allows its users located across the world to chat, based on the type of emergency, but having the chat be location-based, i.e., only allowing users a certain distance away from each other to chat, as they would likely be encountering the same emergency, would be more effective and practical, making the ad-hoc networks of users communicating with each other based on location information.

Moreover, the mobile application could also include the additional feature of alerting authorities when surrounding conditions to a mobile device attain a level higher than a specified safe threshold. For instance, if a mobile device allows the continuous detection of heat and the degree is abnormally elevated, then the mobile application would conclude that a fire has occurred, alert the authorities, and display the fire emergency plan. This is similar to previous work where experts are informed if detected radioactivity levels are high at a given location such that they could assess the risks and apply the procedure suitable for the emergency at hand [17]. This allows emergency procedures for crisis incidents to be revealed automatically, thereby reducing the amount of time needed to

manage the situation which would otherwise take longer due to the limitations in terms of the manual interaction with the system.

5.4 Summary

This chapter describes a possible application of the crisis management DSL and the traversal mechanism, which we choose to use as our proof-of-concept implementation, and consists in the development of an Android mobile application allowing its users to handle a disaster by visualizing the step-by-step process to follow during an emergency situation, based on extended TUCMs defined by crisis experts and executed via the mobile application. We present the features already implemented in the mobile application as well as those which could be added in the future to improve it. The next chapter uses this proof-of-concept implementation by executing extended TUCMs corresponding to real life crisis scenarios on the mobile application in order to validate our crisis management DSL's feasibility and our traversal algorithm's correctness.

Chapter 6

Validation

This chapter demonstrates our domain specific software language's (DSL) feasibility and our traversal algorithm's correctness by testing two selected real life crisis scenarios, namely heart attack and fire, on the mobile application described in the previous chapter. In addition, since many Use Case Map (UCM) [6] variants exist, several other tests covering these variants are performed and mentioned briefly.

6.1 Test Case 1: Heart Attack

The first UCM we test is that of a heart attack emergency situation. Since medical emergencies were not part of the preliminary research we conducted regarding crisis situations, this heart attack test case also serves as an indicator for the generalizability of our proposed crisis management DSL to cover other types of emergencies. Fig. 49 shows the UCM we create for a first responder to deal with this crisis incident, and Listing 4 depicts its equivalent Textual Use Case Map (TUCM) [1]. We provide screenshots of the mobile application when certain important steps in the model are attained, in order to show that our traversal algorithm works properly. To begin, we must select the heart attack emergency type from the crisis situations provided in the mobile application, by choosing the "Medical" icon, followed by the "Heart Attack" option from the displayed dropdown menu, as demonstrated in Fig. 50a. The authority stored in the mobile application is then sent the initial location of the user, in this case the first responder, corresponding to the location denoted at the top of this actor's swimlane in the UCM. Thereafter, the start of the crisis management procedure is revealed, as shown in Fig. 50b. Pressing the "Next" button to go to the next step should lead to the first OrFork in the heart attack UCM, which is

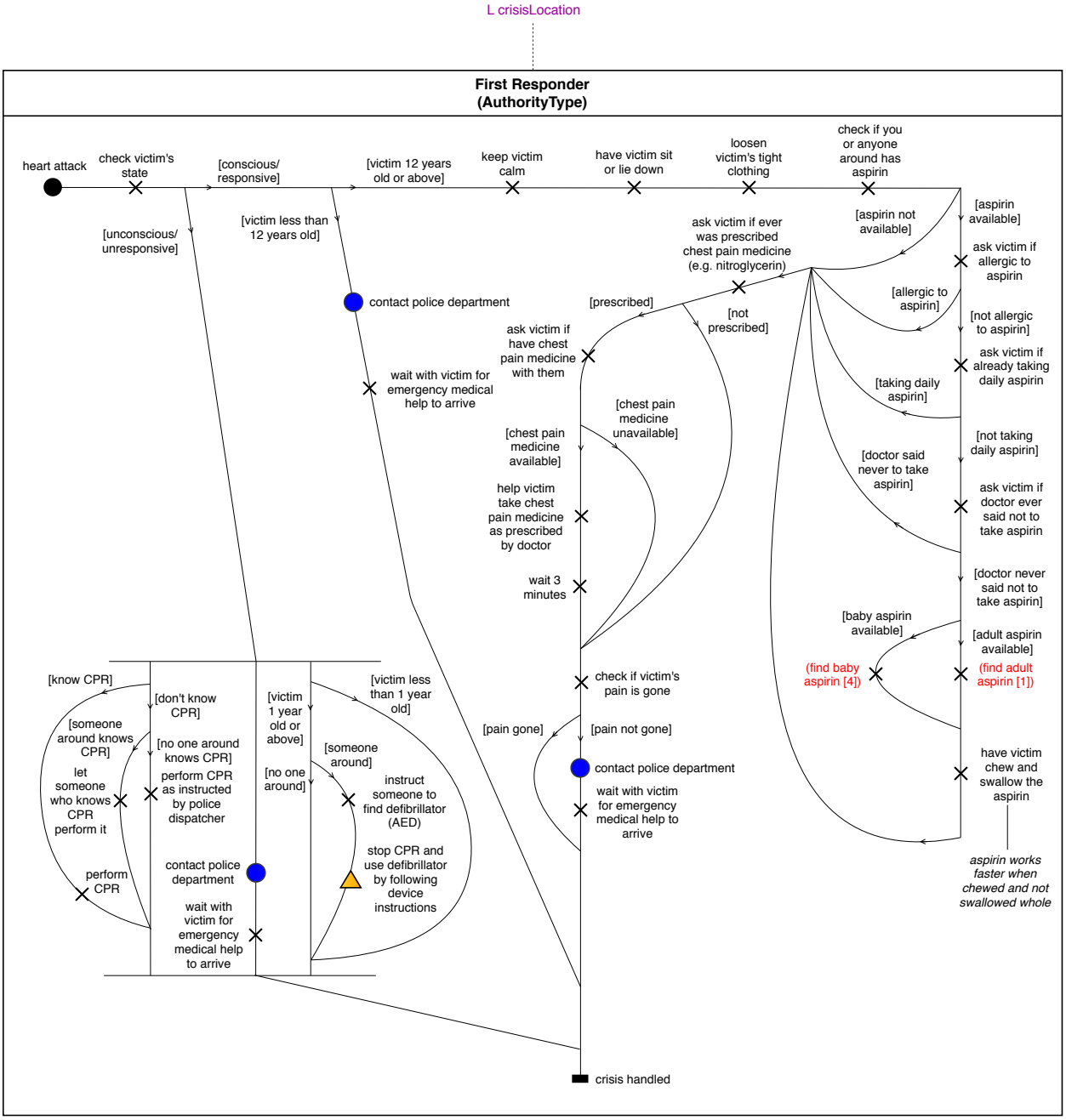


Figure 49: Example UCM for heart attack medical emergency tailored to first responder

indeed the case as a dialog is displayed with a dropdown containing the conditions of the OrFork's branches (see Fig. 51a).

Listing 4:
TUCM corresponding to the heart attack UCM in Fig. 49

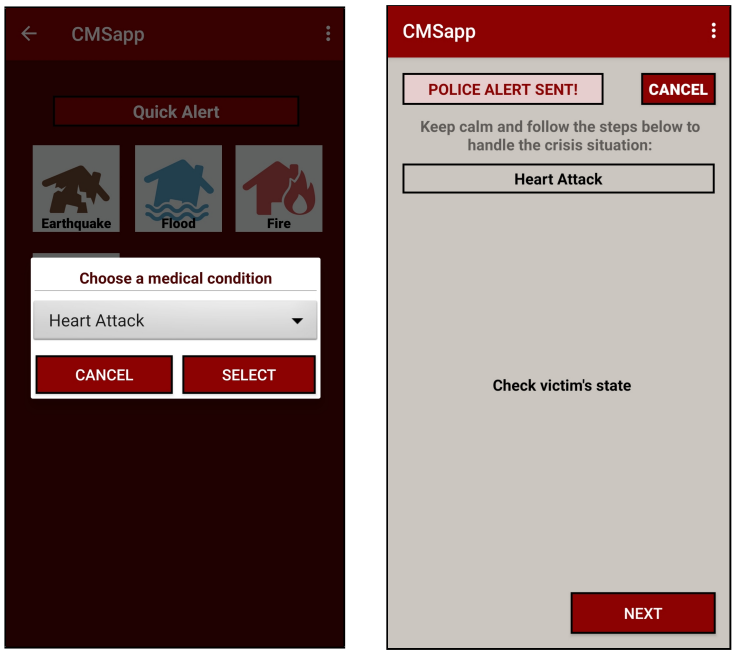
```
urnModel HeartAttackEmergencyType

ResourceType r1#"baby aspirin" locatable
ResourceType r2#"adult aspirin" locatable

map HeartAttackFirstResponder{
start s0#"heart attack"
 -> S s1#"Check victim's state"
 -> or{["conscious/responsive"]
         -> or{["victim 12 years old or above"]
                 -> S s2#"Keep victim calm"
                 -> S s3#"Have victim sit or lie down"
                 -> S s4#"Loosen victim's tight clothing"
                 -> S s5#"Check if you or anyone around has aspirin"
                 -> or{[aspirin not available] -> ;
                       [aspirin available]
                         -> S s6#"Ask victim if allergic to aspirin"
                         -> or{[allergic to aspirin] -> ;
                               [not allergic to aspirin]
                                   -> S s7#"Ask victim if already taking daily
                                      ↪  aspirin"
                                   -> or{[taking daily aspirin] -> ;
                                         [not taking daily aspirin]
                                           -> S s8#"Ask victim if doctor ever
                                              ↪ said not to take aspirin"
                                           -> or{[doctor said never to take
                                              ↪ aspirin]
                                                -> ;
                                              [doctor never said not to
                                                 ↪ take aspirin]
                                                -> or{[baby aspirin
                                                   ↪ available]
                                                     -> R s9#""
                                                        ↪ ResourceType
                                                        ↪ (r1,4)
                                                     -> ;
                                                   [adult aspirin
                                                      ↪ available]
                                                     -> R s10#""
                                                        ↪ ResourceType
                                                        ↪ (r2,1)
                                                     -> ;
                                                }
                                                -> S s11#"Have victim chew
                                                   ↪ and swallow the
                                                   ↪ aspirin" Fact {"
```

```
                                                    ↪ aspirin works faster
                                                    ↪  when chewed and not
                                                    ↪  swallowed whole"}
35                                                -> ;
36                                            } -> ;
37                                        } -> ;
38                                    } -> ;
39                            }
40                        -> S s12#"Ask victim if ever was prescribed chest pain
                             ↪ medicine (e.g. nitroglycerin)"
41                        -> or{[not prescribed] -> ;
42                              [prescribed]
43                                  -> S s13#"Ask victim if have chest pain medicine
                                     ↪ with them"
44                                  -> or{[chest pain medicine unavailable] -> ;
45                                        [chest pain medicine available]
46                                            -> S s14#"Help victim take chest pain
                                               ↪ medicine as prescribed by doctor"
47                                            -> S s15#"Wait 3 minutes"
48                                            -> ;
49                                    } -> ;
50                            }
51                        -> S s16#"Check if victim's pain is gone"
52                        -> or{[pain gone] -> ;
53                              [pain not gone]
54                                -> C s17#"" ActorRoleType(g1)
55                                -> S s18#"Wait with victim for emergency medical help
                                   ↪  to arrive"
56                                -> ;
57                            } -> ;
58                    ["victim less than 12 years old"]
59                      -> C s19#"" ActorRoleType(g1)
60                      -> S s20#"Wait with victim for emergency medical help to
                         ↪ arrive"
61                      -> ;
62                } -> ;
63            ["unconscious/unresponsive"]
64               -> and { * -> C s21#"" ActorRoleType(g1)
65                         -> S s22#"Wait with victim for emergency medical help to
                            ↪  arrive"
66                         -> ;
67                       * -> or{["know CPR"] -> S s23#"Perform CPR" -> ;
68                               ["don't know CPR"]
69                                   -> or{[someone around knows CPR]
70                                             -> S s24#"Let someone who knows CPR
                                                ↪ perform it"
71                                             -> ;
72                                         [no one around knows CPR]
73                                             -> S s25#"Perform CPR as instructed
                                                ↪ by police dispatcher"
74                                             -> ;
75                                       } -> ;
76                               }
77                       * -> or{["victim less than 1 year old"] -> ;
```

```
78                                ["victim 1 year old or above"]
79                                    -> or{[no one around] -> ;
80                                          [someone around]
81                                             -> S s26#"Instruct someone to find
                                                  ↪ defibrillator (AED)"
82                                             -> W s27#"Stop CPR and use
                                                  ↪ defibrillator by following
                                                  ↪ device instructions"
83                                             -> ;
84                                       } -> ;
85                                }
86                    } -> ;
87        }
88     -> end s28#"crisis handled".
89
90    GroupType g1#"police department" ActorRoleType(a1)
91    authorityType a1#"police"
92    }
```

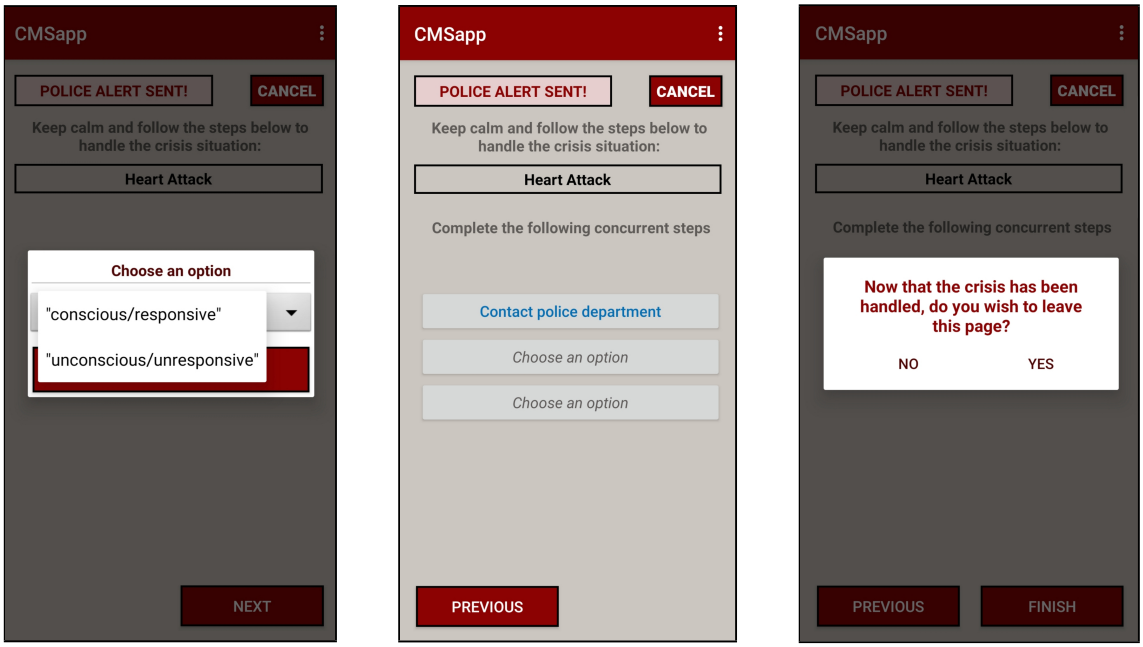
In the event the user selects the "unconscious/unresponsive" branch condition, the AndFork is correctly displayed, as shown in Fig. 51b. Once all steps in the AndFork are complete, the "Next" button becomes a "Finish" button, which we press to send the authority a text message stating that the crisis has been handled (see Fig. 51c). As for the actual AndFork traversal, Fig. 52a depicts one of the OrForks in the AndFork, Fig. 52b shows the AndFork's state after a few accomplished steps, and Figs. 52c and 52d demonstrate the proper display of the CommunicationStep and WarningStep, respectively, in the AndFork. Hence, with all the aforementioned AndFork traversal checks, we confirm that our traversal algorithm is correct when handling AndForks starting or ending with OrForks, containing nested OrForks, and encompassing varying step types, and models having an AndFork as the last path element before their EndPoint.

On the other hand, if the user selects the "conscious/responsive" branch condition, the next OrFork automatically appears, as illustrated in Fig. 53a, and when the user selects the "victims 12 years old or above" branch condition, the step that follows it is displayed, as shown in Fig. 53b. Stepping through the right branches of the OrForks to the right of the UCM, we eventually reach the OrFork in Fig. 54a, and choosing the "adult aspirin available" branch condition leads to the display of the ResourceStep in Fig. 54b. The next step is then the ConnectingOrBody of this OrFork, which includes a Fact, as shown in



(a) (b)

Figure 50: Upon (a) selecting the "Heart Attack" option from the "Medical" emergency dropdown, (b) the crisis management procedure for the selected crisis situation is revealed



(a) (b) (c)

Figure 51: When the "unconscious/unresponsive" branch condition is selected from the OrFork in (a), the AndFork (b) is displayed, and (c) when its steps are complete, the user can end the emergency plan

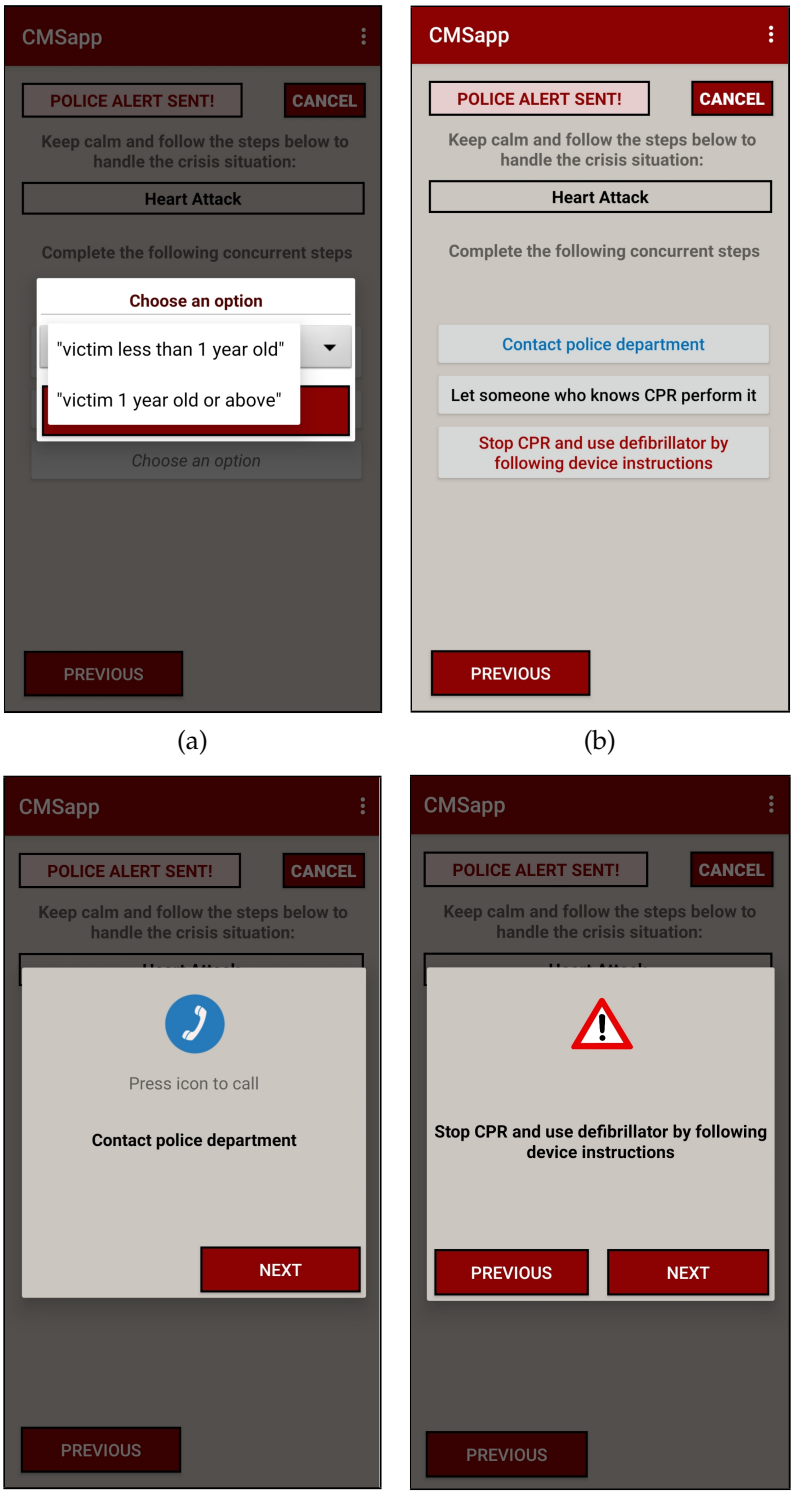
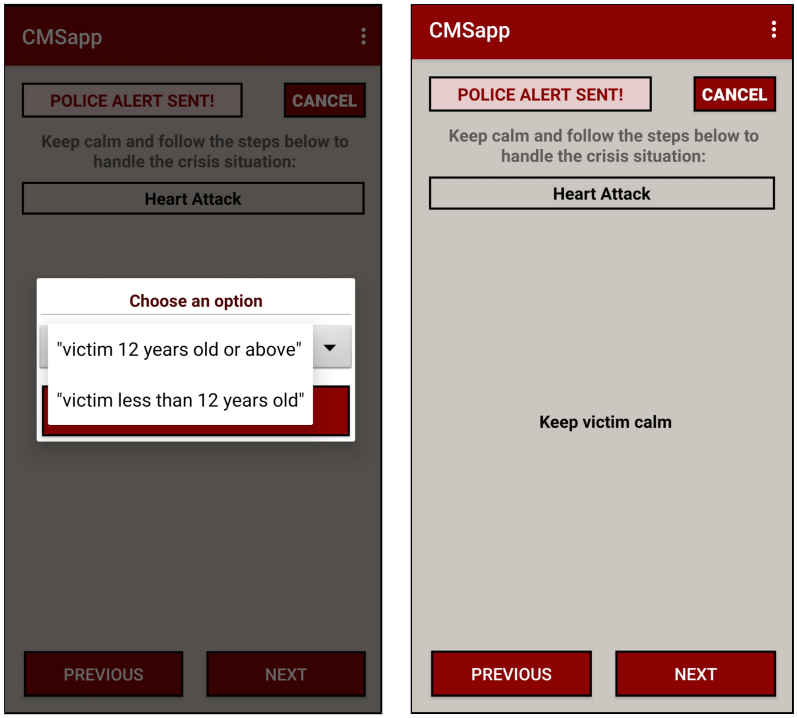
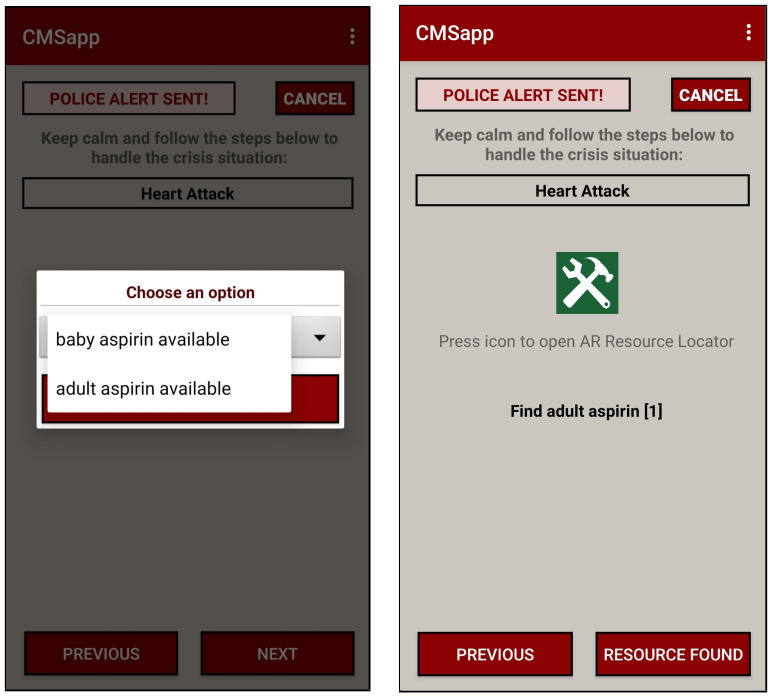


Figure 52: Traversing an AndFork, which (a) may include OrForks, (b) leads to new steps being displayed as the user progresses through the AndFork, and (c)-(d) requires handling varying step types



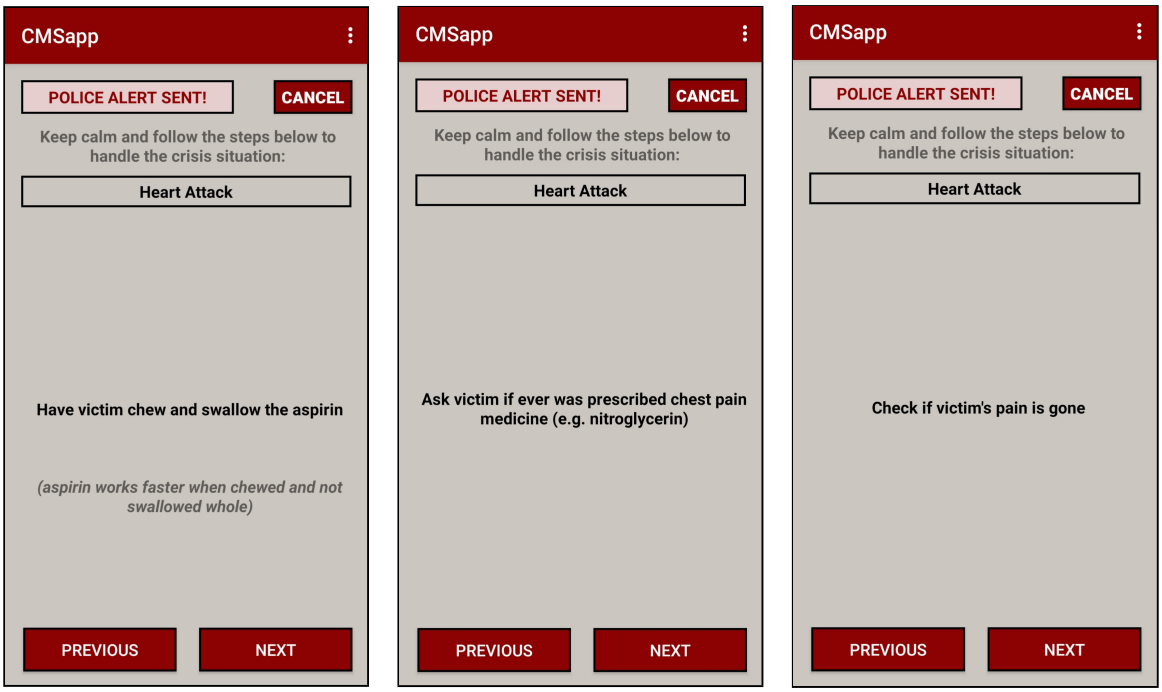
(a) (b)

Figure 53: When the "victim 12 years old or above" branch condition is selected from the OrFork in (a), the step (b) is displayed



(a) (b)

Figure 54: When the "adult aspirin available" branch condition is selected from the OrFork in (a), the ResourceStep (b) is displayed



(a) (b) (c)

Figure 55: ConnectingOrBodies in the heart attack UCM

Fig. 55a. Then, the step following the latter is another ConnectingOrBody but that of the first OrFork in the set of OrForks to the right of the UCM, as shown in Fig. 55b, revealing that our algorithm properly searches the ConnectingOrBody of previous OrForks in which an OrFork is nested. Similarly, choosing the left branches of any of the OrForks to the right of the UCM, apart from the OrFork containing the ResourceSteps, would also yield the step in Fig. 55b. Afterwards, we again have a set of nested OrForks, which have the step in Fig. 55c as ConnectingOrBody. Clicking the "Next" button from this step leads to the OrFork in Fig. 56a. Selecting the "pain is gone" branch condition entails the end of the emergency procedure, which is correctly portrayed in the mobile application by the "Finish" button as shown in Fig. 56b. Conversely, selecting the "pain is not gone" option leads to the step in Fig. 56c followed by that in Fig. 56d, which is the last step of the emergency plan and thus has a "Finish" button. Note that choosing the "victims is less than 12 years old" branch condition at first leads to the same two aforementioned steps being displayed in the same order. Therefore, our traversal algorithm correctly deals with UCMs having an OrFork with an empty branch as the last path element before their EndPoint,

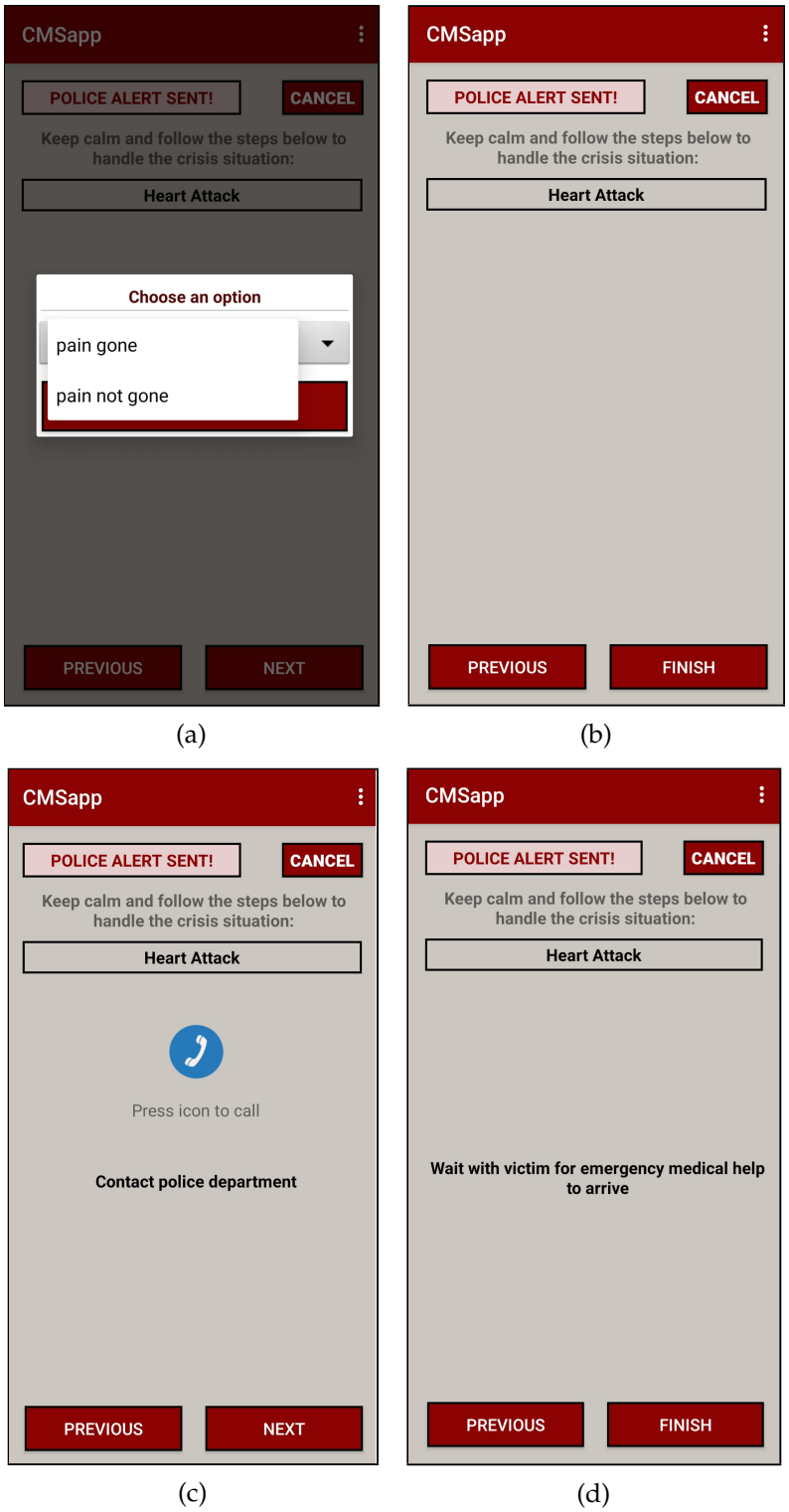


Figure 56: When the "pain gone" branch condition is selected from the OrFork in (a), we reach (b) the end of the UCM, and when the "pain not gone" branch condition is chosen, the CommunicationStep (c) is displayed followed by the last step (d) in the UCM

nested OrForks, and different step types.

6.2 Test Case 2: Fire

Since the heart attack emergency plan does not encompass location changes, we include the fire UCM in Fig. 57 and its equivalent TUCM in Listing 5 to demonstrate that location updates are also handled properly by our traversal algorithm. This fire UCM is essentially the same as the one from Section 3.2.2 but, since the mobile application solely deals with the roles of victim and first responder, the crisis response for the fire scenario provided in this chapter is tailored to the victim role. Fig. 58 presents the five location updates within the fire crisis procedure in order, demonstrated by the "Location Update" and "Resource Found" buttons. When clicked, the former sends a text message to the authority specified in the mobile application with a name designating the location and the current location of

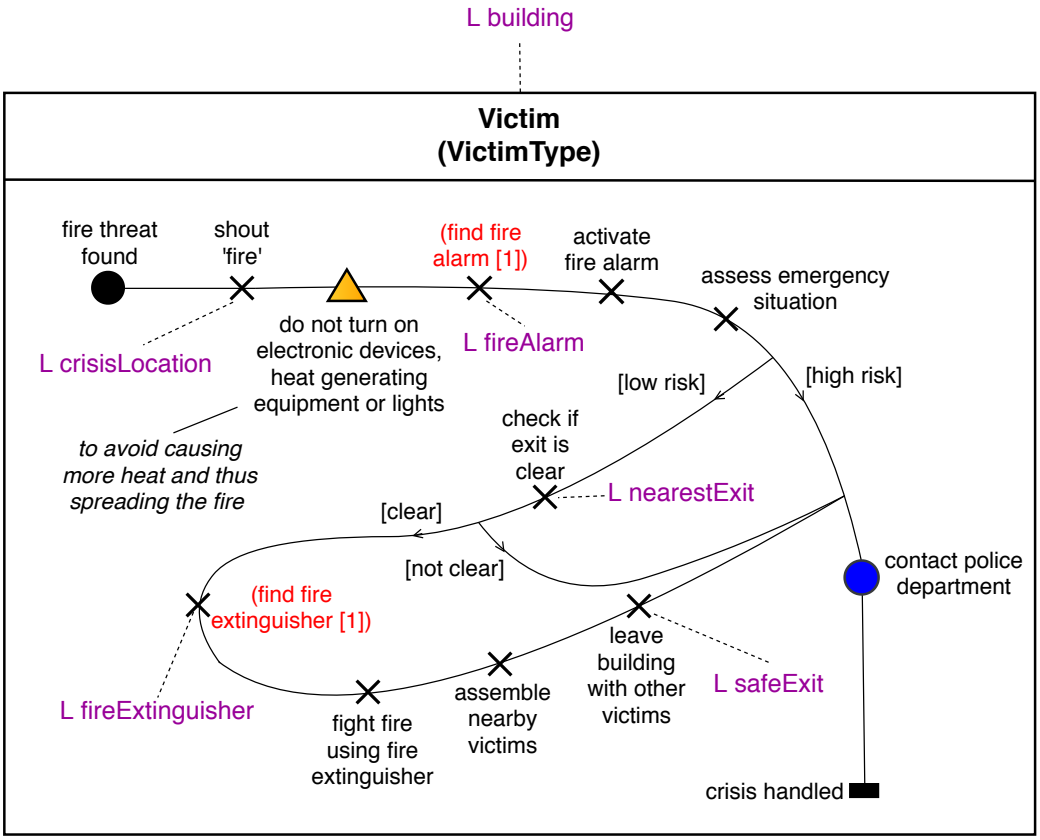


Figure 57: Example UCM for fire emergency tailored to victim

the user, while the latter sends the location of the resource found by the user in addition to the resource's name and quantity. As such, our traversal algorithm properly handles UCMs with location changes and paths outside of AndForks or OrForks (in Fig. 57, the path whose first step is "shout 'Fire'" and last step is "assess emergency situation", and the path containing the "contact police department" step correspond to such paths).

Listing 5:
TUCM corresponding to the fire UCM in Fig. 57

```
urnModel FireEmergencyType

LocationType l1#"crisis location"
LocationType l2#"fire alarm"
LocationType l3#"nearest exit"
LocationType l4#"fire extinguisher"
LocationType l5#"safe exit"

ResourceType r1#"fire alarm" locatable
ResourceType r2#"fire extinguisher" locatable

map FireVictim {
start s0#"fire threat found"
  -> S s1#"Shout 'fire'" LocationType(l1)
  -> W s2#"Do not turn on electronic devices, heat generating equipment or
     ↪ lights" Fact {"to avoid causing more heat and thus spreading the fire
     ↪ "}
  -> R s3#"" LocationType(l2) ResourceType(r1,1)
  -> S s4#"Activate fire alarm"
  -> S s5#"Assess emergency situation"
  -> or { [high risk] -> ;
          [low risk] -> S s6#"Check if exit is clear" LocationType(l3)
                     -> or { [not clear] -> ;
                             [clear] -> R s7#"" LocationType(l4) ResourceType(
                                ↪ r2,1)
                                  -> S s8#"Fight fire using fire
                                     ↪ extinguisher"
                                  -> S s9#"Assemble nearby victims"
                                  -> S s10#"Leave building with other
                                     ↪ victims" LocationType(l5)
                                  -> ;
                           }
                     -> ;
     }
  -> C s11#"" ActorRoleType(g1)
  -> end s12#"crisis handled".

GroupType g1#"police department" ActorRoleType(a1)
authorityType a1#"police"
}
```

Figure 58: Steps in the fire UCM involving actor location changes

6.3 Additional Test Cases

As UCMs may be very different from one another depending on the different path elements they contain and their order, the two aforementioned test cases do not cover all possible UCM variations. Thus, in this section, we present other UCMs we use to test our DSL and traversal algorithm, in order to validate that they in fact work properly.

One test case involves a UCM containing AndForks separated by a step or occurring one after the other (see Fig. 59a) and nested within a model's AndFork (see Fig. 60a). We check both within an AndFork and outside of any AndFork to cover all occurrences of this case in the model, since our traversal mechanism isolates the part of the algorithm dealing with the navigation of AndForks from that of the rest of the elements on a model's Path. Similarly, we test a UCM containing OrForks separated by a step or occurring one after the other (see Fig. 59b) and nested within an AndFork (see Fig. 60b). Moreover, the heart attack test case already demonstrates the correct functionality of our traversal algorithm with respect to a UCM having an OrFork or an AndFork as the last path element before its EndPoint, but we also test a UCM where the first path element after its StartPoint is either an OrFork or an AndFork (see Fig. 59c). Likewise, we test AndFork branches starting and ending with AndForks (see Fig. 60c). Furthermore, we test nested AndForks, with ConnectingAndBodies, to verify that the correct ConnectingAndBody is displayed when the steps preceding it are complete (see Fig. 60d). We also test OrForks and their ConnectingOrBodies in an AndFork (see Fig. 60e). Additionally, we test OrForks containing AndForks (see Figs. 59d and 60f), and OrForks and AndForks occurring one after the other with or without a step separating them (see Figs. 59e and 60g).

So far, we have discussed the behavior of our traversal algorithm when navigating forward in an emergency plan. However, we also test iterating backward and conclude that our traversal algorithm correctly navigates backward in a crisis procedure.

Note that the path, AndForks, and OrForks in the various tested UCMs have a varying number of PathBodyNodes, AndFork branches, and OrFork branches, respectively, to ensure that iterating forward and backward works correctly regardless of the number of

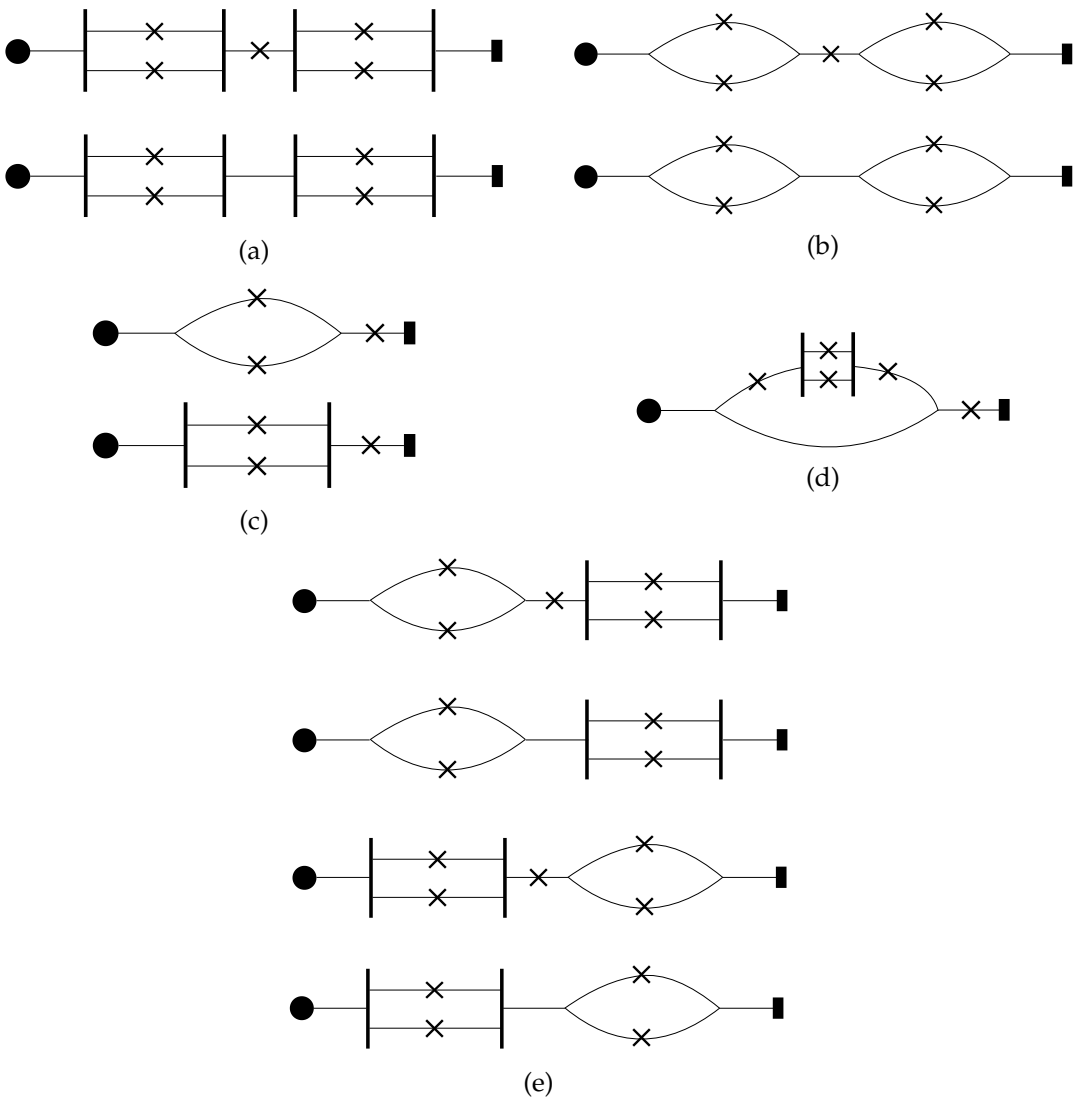


Figure 59: Additional test cases for traversal of UCM Paths containing (a) AndForks separated by a step or occurring one after the other, (b) OrForks separated by a step or occurring one after the other, (d) OrForks encompassing AndForks, or (e) OrForks and AndForks occurring one after the other with or without a step separating them, as well as Paths having (c) an OrFork or an AndFork as first path element following their StartPoint

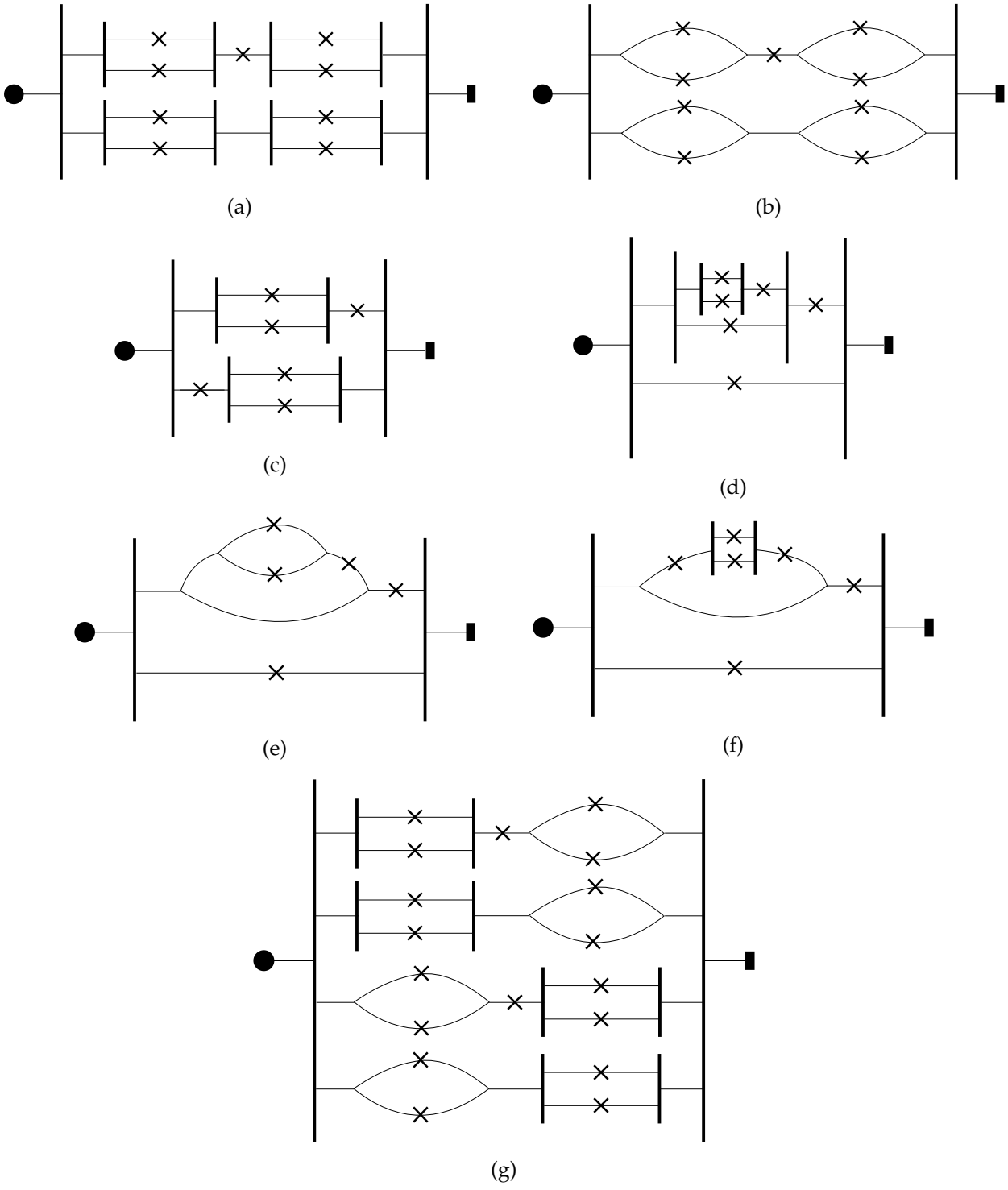


Figure 60: Additional test cases for traversal of UCM AndForks with branches containing (a) AndForks separated by a step or occurring one after the other, (b) OrForks separated by a step or occurring one after the other, (d) nested AndForks with their ConnectingAndBodies, (e) nested OrForks with their ConnectingOrBodies, (f) OrForks encompassing AndForks, or (g) OrForks and AndForks occurring one after the other with or without or without a step separating them, as well as AndFork branches (c) starting and ending with AndForks

PathBodyNodes in a path, AndFork branches in an AndFork, or OrFork branches in an OrFork, respectively.

6.4 Summary

This chapter serves the purpose of validating the feasibility of our crisis management DSL and the correctness of our traversal algorithm. After executing two substantial crisis scenarios on our mobile application, in addition to several smaller UCMs dealing with specific possible UCM variations, we are able to conclude that our DSL and traversal algorithm are indeed performing as intended. The next chapter provides past work related to our present work, to demonstrate our contributions to the field of crisis management systems and workflow modeling.

Chapter 7

Related Work

Previous work involves systems described by different workflow notations. While some systems use the metamodel of a workflow language to model basic procedures, others extend the metamodel with concepts specific to the domain at hand, as is accomplished in the current thesis. For instance, researchers describe the business process in production systems by extending the Unified Modeling Language (UML) [2] Activity Diagram notation with structural and functional concepts, such as activities and resources, related to enterprise workflows and essential to their execution [26]. This extension demonstrates the application of UML to business modeling in addition to the common adoption of UML use cases for describing business workflows [26]. Another example mentions the UML Extended Workflow Metamodel (UML-EWM), which also extends the standard UML Activity Diagram metamodel to model workflow processes [22]. Furthermore, an extension to the Use Case Map (UCM) [6] notation with concepts for task, dialogs, and layout supports the modeling of user interface requirements [13]. Many projects also focus on specific extensions to the Business Process Model and Notation (BPMN) [5] [7]. Some researchers extend BPMN to model business process security requirements by integrating these into the metamodel of Business Process Diagrams (BPDs) [5], thereby forming a domain-specific language (DSL) [12, 20]. Another extension to the BPMN metamodel is introduced to model and visualize the resource requirements of a business process [38]. Moreover, many researchers explore general-purpose language (GPL) extensions to model medical treatment processes [11].

Furthermore, past work discusses software applications, such as simulation tools, to execute workflows [27]. Process simulation is different from real process execution, but often lies in the execution environment [7]. The execution of a workflow refers to the automatic

retrieval of the next step to follow in a process upon the completion of the step or steps that precede it, while supporting concurrent and alternative paths [4]. Languages, such as the Web Services - Business Process Execution Language (WS-BPEL) [39, 40], are designed to allow the execution of business processes [38]. However, as WS-BPEL has limitations in terms of what aspects of BPMN diagrams it can represent, a standard eXtensible Markup Language (XML) [41] interchange format for BPMN diagrams, named XML Process Definition Language (XPDL), is created to capture all information in such diagrams in order to validate their correctness and simulate the execution of business processes [7]. Previous work also devises execution algorithms for UML state machines [42] and activity graphs [43]. As for the User Requirements Notation (URN) [6], the jUCMNav Eclipse plugin allows the execution of models created via the Goal-oriented Requirement Language (GRL) and UCM notations using the Strategies and Scenarios views, respectively [44]. However, the aforementioned tool does not dynamically execute UCMs, as the path followed in a UCM is based on the path variable values set in a scenario definition created prior to executing it on a model [45].

With respect to crisis management, growing attention is given to modeling and simulating emergency procedures in order to analyze and test them [23]. We can use a workflow management system to execute a given workflow in order to automate emergency plans [46]. Past work builds simulation tools for crisis scenarios based on the Crisis and Emergency Modeling Language (CEML), a graphical modeling language specializing a subset of SysML to the crisis management domain [23]. CEML is used by domain experts to create models of emergency scenarios which are then converted into simulation models [23]. The simulation tools use the metamodel of CEML, depicting the structural aspects of crisis scenarios, and collaboration design patterns focusing on the interactions and communication between actors involved in a crisis [23]. Another example combines workflow and rule-based languages to model flexible emergency plans, as crisis procedures must sometimes be adapted to the occurrence of complex and unexpected situations [47].

Although various domains are described using the aforementioned extended notations, none specifically focuses - to the best of our knowledge - on the crisis management domain,

takes the 3D environment into account, and considers modern social media like interactions. In addition, no previous work is found that explores the real-time execution of UCMs. Note that, while any of the aforementioned existing workflow notations could be used for the purposes of this thesis, we choose to extend URN due to our familiarity with this notation.

Chapter 8

Conclusion and Future Work

This thesis presents a crisis management domain-specific software language (DSL) exemplified by an extension of the User Requirements Notation's (URN) [6] Use Case Map (UCM) notation that incorporates crisis management concepts into the UCM metamodel, as a first step toward the long-term goal of providing society with a more efficient way of dealing with emergency situations. We determine the requirements for this DSL based on existing crisis procedures, focusing on natural disasters, specifically earthquakes, floods, and fires. Concrete syntax elements for the added crisis management concepts are included into the existing UCM representation to highlight resources needed during an emergency situation, facts that victims and authorities should be aware of, actors communicating among each other, warnings that victims and authorities should be attentive to, and the locations of emergencies, and their associated actors and resources. While the UCM notation is used in this thesis, the discussed crisis management concepts may similarly be applied to other workflow notations.

Although existing workflow notations can model basic emergency procedures, they do not focus on crisis management systems, consider the 3D environment in which crisis events unfold, and take social media like interactions into account. The proposed crisis management DSL captures these key aspects and hence supports modeling location-based, social media-inspired interactions in the 3D environment of a disaster.

Once created, our DSL, which specifically extends the textual implementation of UCM (TUCM) offered by the Textual User Requirements Notation (TURN) [1] with crisis management concepts, provides a crisis authoring environment used by crisis experts to author emergency procedures. Their created crisis response models are subsequently traversed using our algorithm allowing the forward and backward navigation of a selected

model. Our crisis management DSL is then used as the foundation for an Android mobile application we build for victims and first responders which reads the crisis response models, created using our DSL, and uses our traversal algorithm to step through them. The mobile application is our proof-of-concept implementation, on which we are able to execute real life emergency scenarios, in order to validate the feasibility of our DSL and the correctness of our traversal algorithm. From testing various TUCMs on the mobile application, we conclude that indeed, our DSL is feasible and our traversal algorithm is correct. In addition, our mobile application uses modern social media ad-hoc networks for victims and first responders to communicate with each other, rather than the traditional one-on-one voice only 911 calls, thereby making crisis management more effective.

For future work, we wish to improve our crisis management mobile application, as detailed in Section 5.3, and briefly described as follows. First, to more effectively use resources, we want to add augmented reality features to our mobile application to direct users toward locatable resources during a crisis. We also would like our mobile application to allow users to be informed of emergency procedures even while not being exposed to any crisis situation. In addition, we wish to finish implementing our push notifications feature to let users know about reported disasters nearby or new received chat messages. Another improvement involves the possibility of saving more than one contact phone number in our mobile application so that a user may reach a specific victim or authority in parts of an emergency plan in which these actors are relevant. Moreover, the mobile application could handle more than one role type by allowing users to select their role type at the same time as the type of emergency at the start of the mobile application, in order to display the procedure tailored to their specific role. We also wish to include a prompt upon the cancellation of a crisis alert asking the user to provide the reason for this action to the authority whose phone number is stored in the mobile application. Furthermore, updating the chat to being location-based so that users likely encountering the same emergency at a given location may communicate with each other would make the mobile application's chat feature more practical. A final improvement to the mobile application involves the feature of alerting authorities and displaying emergency plans

automatically when a mobile device's surrounding conditions are deemed unsafe. For example, when a device detects heat levels higher than a specified safe threshold, the mobile application could conclude that the latter are due to a fire emergency, and thus directly alert the authorities and display the fire emergency procedure.

Moreover, although we were able to deal with problematic situations encountered while implementing our traversal algorithm by adding empty PathBodyNodes within AndForks and before an AndFork representing the first path element after a UCM's StartPoint, we would like to improve the strategy we used to resolve these issues in the future. We would also like to employ a more efficient way of using the doubly linked list data structure in our traversal mechanism implementation.

Lastly, we plan to do an empirical study by providing crisis victims and first responders with our mobile application, incorporating the emergency plans authored by crisis experts, so that they may verify that the mobile application serves its purpose of properly helping society in managing crisis situations.

References

[1] R. Kumar, "Textual User Requirements Notation," Master's thesis, McGill University, Montreal, 2018.

[2] Object Management Group (OMG), "Welcome To UML Web Site!," [Online]. Available: http://www.uml.org/.

[3] J. Lee and T. Bui, "A Template-based Methodology for Disaster Management Information Systems," in *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, p. 7, IEEE, 2000.

[4] C. Sell and I. Braun, "Using a Workflow Management System to Manage Emergency Plans," in *Proceedings of the 6th International ISCRAM Conference*, vol. 41, p. 43, 2009.

[5] Object Management Group (OMG), "Business Process Model & Notation™ (BPMN™)," *About the Business Process Model and Notation Specification Version 2.0.2*, 2014. [Online]. Available: http://www.omg.org/bpmn/.

[6] International Telecommunication Union (ITU), "Recommendation Z.151 (10/18): User Requirements Notation (URN) - Language definition," 2018. [Online]. Available: https://www.itu.int/rec/T-REC-Z.151/en.

[7] M. Chinosi and A. Trombetta, "BPMN: An introduction to the standard," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 124–134, 2012.

[8] A. Basu and A. Kumar, "Research Commentary: Workflow Management Issues in e-Business," *Information Systems Research*, vol. 13, no. 1, pp. 1–14, 2002.

[9] R. Braun and W. Esswein, "Classification of Domain-Specific BPMN Extensions," in *IFIP Working Conference on The Practice of Enterprise Modeling*, pp. 42–57, Springer, 2014.

[10] M. Strembeck and U. Zdun, "An Approach for the Systematic Development of Domain-Specific Languages," *Software: Practice and Experience*, vol. 39, no. 15, pp. 1253–1292, 2009.

[11] R. Braun, H. Schlieter, M. Burwitz, and W. Esswein, "Extending a Business Process Modeling Language for Domain-Specific Adaptation in Healthcare," in *Proceedings of the 12th International Conference on Wirtschaftsinformatik*, pp. 468–481, 2015.

[12] A. Rodríguez, E. Fernández-Medina, and M. Piattini, "A BPMN Extension for the Modeling of Security Requirements in Business Processes," *IEICE Transactions on Information and Systems*, vol. 90, no. 4, pp. 745–752, 2007.

[13] D. Amyot and G. Mussbacher, "User Requirements Notation: The First Ten Years, The Next Ten Years," *JSW*, vol. 6, no. 5, pp. 747–768, 2011.

[14] N. B. Khzam and G. Mussbacher, "Domain-Specific Software Language for Crisis Management Systems," in *2018 IEEE 8th International Model-Driven Requirements Engineering Workshop (MoDRE)*, pp. 36–45, IEEE, 2018.

[15] G. Negash, C. M. Liang, F. Al Taha, N. B. Khzam, and G. Mussbacher, "Non-Deterministic Use Case Map Traversal Algorithm for Scenario Simulation and Debugging," in *2019 IEEE 9th International Model-Driven Requirements Engineering Workshop (MoDRE)*, IEEE, 2019. (to be published).

[16] M. Avvenuti, S. Cresci, A. Marchetti, C. Meletti, and M. Tesconi, "EARS (Earthquake Alert and Report System): a Real Time Decision Support System for Earthquake Crisis Management," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1749–1758, ACM, 2014.

[17] H.-Y. Mak, A. P. Mallard, T. Bui, and G. Au, "Building online crisis management support using workflow systems," *Decision Support Systems*, vol. 25, no. 3, pp. 209–224, 1999.

[18] J. Kienzle, N. Guelfi, and S. Mustafiz, "Crisis Management Systems: A Case Study for Aspect-Oriented Modeling," in *Transactions on Aspect-Oriented Software Development VII*, pp. 1–22, Springer, 2010.

[19] A. Van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, 2000.

[20] M. Saleem, J. Jaafar, and M. Hassan, "A Domain-Specific Language for Modelling Security Objectives in a Business Process Models of SOA Applications," *International Journal on Advances in Information Sciences and Service Sciences*, vol. 4, no. 1, pp. 353–362, 2012.

[21] M. Rosemann and M. Zur Muehlen, "Evaluation of workflow management systems - a meta model approach," *Australasian Journal of Information Systems*, vol. 6, no. 1, 1998.

[22] N. Debnath, D. Riesco, M. P. Cota, J. G. Perez-Schofield, and D. Uva, "Supporting the SPEM with a UML Extended Workflow Metamodel," in *IEEE International Conference on Computer Systems and Applications*, pp. 1151–1154, Citeseer, 2006.

[23] A. De Nicola, A. Tofani, G. Vicoli, and M. L. Villani, "An MDA-based Approach to Crisis and Emergency Management Modeling," *International Journal on Advances in Intelligent Systems*, vol. 5, pp. 89–100, 2012.

[24] Object Management Group (OMG), "OMG," [Online]. Available: https://www.omg.org/.

[25] International Telecommunication Union (ITU), "ITU," [Online]. Available: https://www.itu.int/.

[26] R. M. Bastos and D. D. A. Ruiz, "Extending UML Activity Diagram for Workflow Modeling in Production Systems," in *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pp. 3786–3795, IEEE, 2002.

[27] W. M. Van Der Aalst, "Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management," in *Advanced Course on Petri Nets*, pp. 1–65, Springer, 2003.

[28] D. Amyot, "Introduction to the User Requirements Notation: Learning by Example," *Computer Networks*, vol. 42, no. 3, pp. 285–301, 2003.

[29] Ready, "Earthquakes," *Department of Homeland Security*. [Online]. Available: https://www.ready.gov/earthquakes.

[30] Government of British Columbia, "B.C. Earthquake Immediate Response Plan," *Public Safety & Emergency Services*. [Online]. Available: https://www2.gov.bc.ca/assets/gov/public-safety-and-emergency-services/emergency-preparedness-response-recovery/provincial-emergency-planning/irp.pdf.

[31] Southeastern Oklahoma State University, "Emergency Preparedness & Crisis Management Plan," *Safety Department*, 2012. [Online]. Available: http://homepages.se.edu/public-safety/files/2009/12/emergency-preparedness-plan12.pdf.

[32] University of Cambridge, "Fire & Emergency Procedure," *Department of Engineering Health & Safety*. [Online]. Available: https://safety.eng.cam.ac.uk/procedures/Emergency/FireEmergency.

[33] Ready, "Floods," *Department of Homeland Security*. [Online]. Available: https://www.ready.gov/floods.

[34] Government of Canada, "Floods - What to do?," *Department of Public Safety and Emergency Preparedness*. [Online]. Available: https://www.getprepared.gc.ca/cnt/rsrcs/pblctns/flds-wtd/index-en.aspx.

[35] Object Management Group (OMG), "About the XML Metadata Interchange Specification Version 2.5.1," 2015. [Online]. Available: https://www.omg.org/spec/XMI/About-XMI/.

[36] Eclipse Foundation, "Eclipse Modeling Framework (EMF)," [Online]. Available: https://www.eclipse.org/modeling/emf/.

[37] A. Ben Fadhel, *Comprehensive Specification and Efficient Enforcement of Role-based Access Control Policies using a Model-driven Approach*. PhD thesis, University of Luxembourg, Luxembourg, 2017.

[38] L. J. R. Stroppi, O. Chiotti, and P. D. Villarreal, "A BPMN 2.0 Extension to Define the Resource Perspective of Business Process Models," in *XIV Congreso Iberoamericano en Software Engineering*, 2011.

[39] M. Owen and J. Raj, "BPMN and Business Process Management," *Introduction to the New Business Process Modeling Standard*, 2003.

[40] OASIS, "OASIS Web Services Business Process Execution Language (WSBPEL) TC," [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[41] World Wide Web Consortium (W3C), "Extensible Markup Language (XML)," [Online]. Available: https://www.w3.org/XML/.

[42] T. Santen and D. Seifert, *Executing UML State Machines*. Citeseer, 2006.

[43] R. Eshuis and R. Wieringa, "An Execution Algorithm for UML Activity Graphs," in *International Conference on the Unified Modeling Language*, pp. 47–61, Springer, 2001.

[44] A. Pourshahid, D. Amyot, L. Peyton, S. Ghanavati, P. Chen, M. Weiss, and A. J. Forster, "Business process management with the user requirements notation," *Electronic Commerce Research*, vol. 9, no. 4, pp. 269–316, 2009.

[45] G. Mussbacher, S. Ghanavati, and D. Amyot, "Modeling and Analysis of URN Goals and Scenarios with jUCMNav," in *2009 17th IEEE International Requirements Engineering Conference*, pp. 383–384, IEEE, 2009.

[46] M. La Rosa and J. Mendling, "Domain-driven Process Adaptation in Emergency Scenarios," in *International Conference on Business Process Management*, pp. 290–297, Springer, 2008.

[47] M. Llavador, P. Letelier, M. C. Penadés, J. H. Canós, M. Borges, and C. Solís, "Precise yet Flexible Specification of Emergency Resolution Procedures," in *3rd International Conference on Information Systems for Crisis Response and Management (ISCRAM)*, 2006.