Interpreting chain of events for safety analysis

Safoura Shamsolketabi

Master in Information Systems
Submission date: June 2013
Supervisor: Tor Stålhane, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU
Norwegian University of
Science and Technology

Interpreting Chains of Events for Safety Analysis

TDT4900 - Information Systems, Master Thesis

Safoura Shamsolketabi

Supervisor: Tor Stålhane

Spring 2013

Abstract

Because of difficulties of safety analysis in large systems and the complexities of managing large amount of data in these systems, the need for a supporting system has become an important area of research. Managing requirements, components and failure modes of a large system for safety analysis without tool support is difficult and could result in skipping or missing details which may cause an accident.

The main goal of this project is to develop a system model for safety analysis facilitation. Sequences of events in a system and its environment may cause an accident in the system's environment. People often have problem following long cause-consequence sequences of events, while accidents with a short path from initiating event to accident are easy to identify. The system model developed in this project enables automatic generation of event sequences that can cause an accident to the environment of a system. For this purpose, the system model uses the domain ontology as its knowledge base. This ontology must contain cause-consequences reported from a safety expert. A rule engine is used for reasoning about these cause-consequence concepts and generating event chains. The process of developing this system model, how the system model uses a domain ontology as its knowledge base and how the cause-consequence concepts should be added to the domain ontology are described in this project. Finally, the developed system model is tested with a real world example (a simplified steam boiler) and the expected event chains generated automatically.

Preface

This report has been written by Safoura Shamsolketabi as part of the subject TDT 4900 – Computer and Information Science, master thesis and was produced during the spring of 2013. It was written at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU).

The purpose of this report is facilitating safety analysis by developing a system model which enables interpreting event chains for a selected domain.

I would like to thank my supervisor Tor Stålhane for great inputs and helpful guidances throughout the project.

Trondheim, 10.06.2013

Safoura Shamsolketabi

Table of contents

List of figures

1 Introduction

This chapter provides some introductory information to the thesis project “Interpreting chain of events for safety analysis”.

1.1 Project background

Safety is a system quality characteristic and a safe system behaves in such a way that it does not harm people, equipment or the environment in which the system is placed. Safety analysis is important and should be performed by people who have experience and knowledge in the related area. As systems grow, analyzing safety requires large amounts of resources. Many components that work together, failure modes and environment effects on the system should be properly controlled, checked and analyzed in order to prevent accidents. These issues clarify the need for a tool which can systematically perform at least part of the safety analysis.

All potentially dangerous events (hazards), the events consequences and its cause should be identified for safety analysis. For this purpose several methods of hazard identification and risk assessment have been used such as FMEA, HazOp and FTA [1]. Currently a tool called DODT [2] provides some safety analysis. Using the DODT tool it is possible to generate part of a FMEA table. The GNLQ [3] tool is a knowledge-based, guided requirements elicitation environment. This tool can also be used for performing safety analysis. GNLQ is developed to semi-automate requirement elicitation process. In GNLQ the combination of ontologies, boilerplates and HazOp templates allow to the tool to provide support for a quality check of the requirements and a functional HazOp of the system, especially in the early phases. In this regard, three concepts have been used – boilerplates, ontologies and generic failure modes. Boilerplates and ontologies are used for requirements analysis and the generic failure modes are added for safety analysis.

My task is to provide a solution which may facilitate semi-automation of safety analysis, at least to some extent. By implementing safety analysis semi-automatically, managing a large set of components would be performed accurately and we will not skip or forget details. In addition, resource consumption will be reduced. Semi-automation of safety analysis also allows people who have limited safety analysis experience to identify hazards in a system. This is especially important in the early phases of a system development.

“Performing hazard identification requires; 1) Knowledge about the domain, 2) Experience and creativity with linking system failures to environment effects and 3) Persistency to follow all effects through from system to environment. These issues cannot be fully automated but a computerized tool can provide assistance” [4].

1.2 Project description

This project is related to the master thesis "Interpreting chains of events for safety analysis", which aims to facilitate safety analysis by reasoning about chains of events and generating event chains for a system. This will be done in the context of a development project to make a realistic prototype for a real-world system. In order to claim that a system is safe, we should be able to identify possible hazards in the system and prevent these hazards from having an unwanted effect on the environment, for example humans or building.

The sequences of dependent events or the event chains describes how the effect of a specific failure is the cause of another failure, for example Figure 1.1 shows the effect of failure A, causes failure B and the effect of failure B, causes failure C. In other word, A causes B and B causes C or, A causes C.

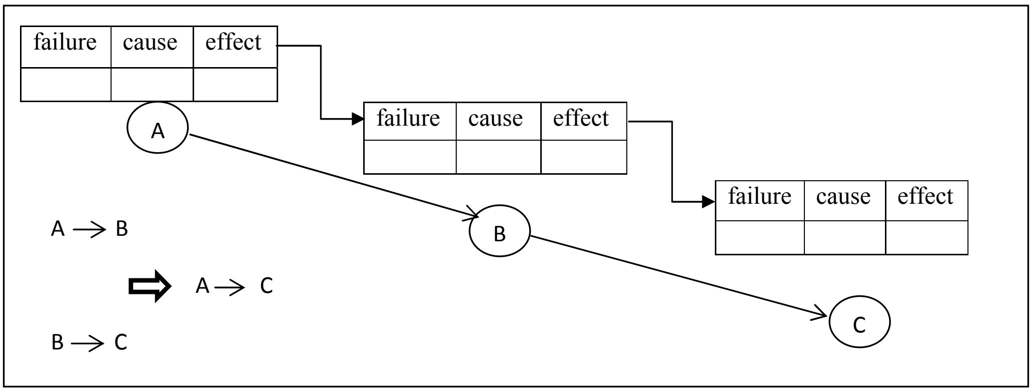


Figure 1.1: Illustration of cause-effect chain

For instance, for a steam boiler, the fire can occur because the hot vessel is located near combustible material. Therefore the cause of fire is the hot vessel and combustible material in the environment. This example shows that in order to be able to identify possible accidents in the system's environment; we need information about the system itself and also information about the environment where a specific system will be placed. Each event chain starts with an initial event and may ends with an accident. For a complex system that contains many components, each with a set of failure modes, following all cause–consequence chains is not doable manually, while accidents identification with a short path from initiating event to accident in a simple system is easy to perform. Figure 1.2 shows a chain of events (A, B and C) causes "Too hot vessel", these events occur in the system. Too hot vessel and combustible material (in the environment) causes a fire.

In order to construct these event chains for a system, we need to have information about the system's environments where the system will be placed. This information can be stored as system or environments ontologies. An ontology formally represents knowledge as a set of concepts within a domain, and the relationships between them. It can be used to model a domain and support reasoning about entities. More details about how ontologies will be used in the process of creating chains of events will be discussed in Chapter 5.

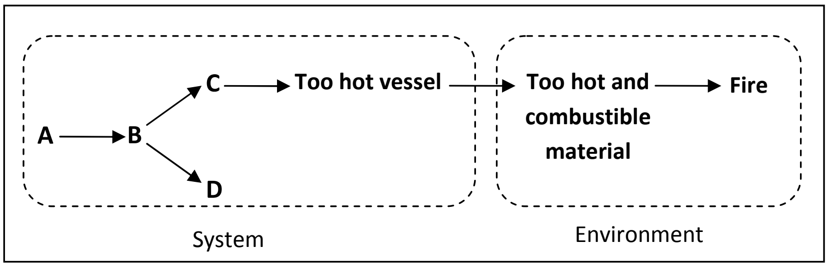


Figure 1.2: Chain of events

The GNLQ is a knowledge-based, guided requirements elicitation environment; this tool also can be used for performing safety analysis. For this purpose GNLQ needs to be upgraded. Currently reasoning about chains of events is not implemented in the tool. Therefore, in this project, I will describe an algorithm for reasoning about chains of events. One idea for describing this algorithm can be to use fault tree analysis. But, conventional fault tree analysis requires that the basic events in the tree occur independently. This is not satisfied when failures occur sequentially.

1.3 Project purpose

Performing safety analysis becomes complex in large systems and small mistakes can cause serious problems. The main goal of this project is to facilitate safety analysis by developing a system model which enables automatic generation of event chains for a defined domain. The developed system, enables generating sequential events in a given domain by using the ontology of this domain as a knowledge base. This ontology contains safety expert knowledge such as cause-consequence chains for the selected domain-information about cause event leading to consequence event:

Cause event → Consequence event

For example:

Event A → Event B
Event B → Event C
Event C → Event D

Using the developed system model, indirect connections will be inferred automatically and finally the complete event chains from the initial event to the final event will be generated:

Event A → Event B → Event C → Event D

The event chains generation system can assist in:

1) (Re) use of safety expert knowledge
2) Improvement of the current safety knowledge
3) Increasing the accuracy of safety analysis
4) Reduction in faults which results from manually safety analysis
5) Reduction in required resources for safety analysis performance

1.4 Research questions

1) Is it possible to develop a system model which enables automatic generation of event sequences for a selected domain?
2) Is it possible to use ontologies as knowledge-base for developing this system?
3) How can we use available methods for developing this system?
4) How can we add cause-consequence concepts to an ontology?

1.5 Project outline

This report is divided into six chapters. Chapter 1 describes the project motivation and background and describes the purpose of this project. Chapter 2 presents the literature study of available methods and technologies for developing an event chains generation system. Chapter 3 discusses the available methods explained in Chapter 2 for selecting an appropriate method in this project. Chapter 4 describes existing systems for supporting "forward chaining" method and provides a discussion on how to select the better system based on a defined set of criteria. Chapter 5 explains the development process of the evens chains generation system. In addition the test description of the developed system against a real world example is provided in this chapter. Chapter 6 gives the conclusions of the project and discusses some features for improving the developed system in further work of this project.

2 Preliminary study

In order to use the information of a system and be able to reason about event chains which occur in the system or its environment, a set of concepts are used. In this chapter, we will describe the introductory concepts for event chain creation.

2.1 What is "Reasoning"?

A set of processes that enables inferring more data from the given data, is called "Reasoning". A system which performs Reasoning is a "Reasoner". A Reasoner uses available information for inference. For example, a system that performs reasoning can use the following data and infer about them:

Given data: **A** is mother of **B**, **B** is mother of **C**
Inferred conclusion: **A** is ancestor of **C**

In this way; reasoning enables us to improve our knowledge of a specific domain. The reasoning process is something that we do all the time, usually unconsciously. But our brain is not so good when reasoning about a large amount of data. If we do that, we will forget and skip data and the result would not be accurate and reasonable. For this reason, getting help from a system that performs reasoning provides large benefits.

A system is able to infer from data if the data is in a machine understandable format. One tool that enables us to store our data in a machine understandable format is Protégé [5]. This tool will be described in Chapter 5. Using Protégé we can develop ontologies and a reasoner can find facts that are implicit in the given ontology. The ontology is explained in the next section.

2.2 What is an ontology?

"An ontology is a description (like a formal specification of a program) of the concepts and relationships that can formally exist for an agent or a community of agents. This definition is consistent with the usage of ontology as set of concept definitions, but more general. And it is a different sense of the word than its use in philosophy." (Gruber 1993)

Using ontology, we can specify concepts of a domain and relationships that exists between concepts, in a formal and machine-understandable format which enables better knowledge management. In addition, an ontology defines a common vocabulary for people who need to share information in a domain. The main reasons why someone needs to develop an ontology are: (1) To share common understanding of the structure of information among people, (2) To enable reuse of domain knowledge, (3) To make domain assumptions explicit and (4) To analyze and improve domain knowledge [6]. Description of how ontologies used in the process of creating event chains, will be provided in Chapter 5.

2.3 Expert systems

Artificial Intelligence (AI) is a wide area of research. The main focus of AI is to make computers reason like people. In Artificial Intelligence, an Expert System is a computer system that uses knowledge representation to enable codifying the knowledge into a knowledge base, which can be used for reasoning. We can process data with this knowledge base to infer conclusions. A knowledge Base (KB) is a special kind of database for knowledge management. A knowledge base is an information repository that provides a means for information to be collected, organized, shared, searched and utilized. Expert Systems are also known as Knowledge-Based Systems.

Expert Systems convert the knowledge of an expert in a specific domain into software. This can be used for answering questions and solving problems. Expert systems usually contain three parts: a knowledge base, an Inference engine and an Interface. The knowledge base contains the information acquired by interviewing experts, and a set of logic rules that govern how that information is applied. The Inference engine interprets the submitted problem against the rules and information stored in the knowledge base. The interface of a knowledge base allows the user to express the problem.

2.4 Rule Engine

A Rule Engine is a system that uses rules, in any format that can be applied to data, to produce a result. A Production Rule System is a kind of Rule Engine and also an Expert System. The advantage of using Rule Engines is that rules can make it easy to express solutions to difficult problems and thus have those solutions verified. Rules are much easier to read than code. Tools such as Eclipse enable editing and managing rules and getting immediate feedback, validation and content assistance [7].

2.4.1 Production System

A Production Rule System uses a knowledge representation to express propositional and first order logic in an easy-to-understand way. The Production Rules System uses an Inference Engine for inferring new data from available data. The Inference Engine is able to manipulate a large number of rules and facts. This engine matches facts and data against Production Rules to infer conclusions which result in actions. A Production Rule has two parts and uses the First Order Logic for reasoning over knowledge representation [7].

- If <conditions> then <actions>

The process of matching facts against Production Rules is called Pattern Matching, which is performed by the Inference Engine. The pattern matching process will be described in the following sections. There are a number of algorithms used for pattern matching by Inference

Engines, but the “Rete” algorithm [8] used in this project. This algorithm will be explained in Section 2.5.

In a Rule Engine, the rules are stored in the Production Memory. The facts that the Inference Engine matches rules against are kept in the Working Memory. Facts are inserted into the working memory, where they may be modified. A system with a large number of rules and facts may result in many rules being true for the same fact; we say that these rules are in conflict. Figure 2.1 represents a high level view of an Inference Engine.

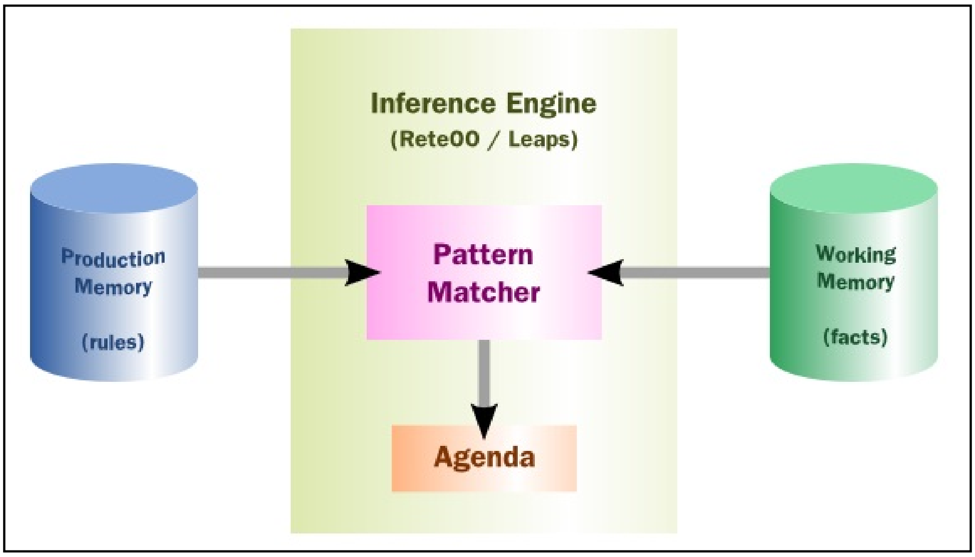


Figure 2.1: High-level view of an inference engine, rules and facts [7]

2.4.2 Forward chaining and backward chaining

There are two methods of execution for a Rule System: Forward Chaining and Backward Chaining. Systems that implement both forward and backward chaining are called Hybrid Chaining Systems. Understanding these two methods of execution help us to understand how a Rule System works. In the following, the two methods together with an example for each method are described.

Forward chaining

The forward chaining method starts with the available data and uses rules to extract more data, until a goal is reached. An inference engine that uses forward chaining, searches the rules until it finds one rule in which the If-clause (condition) is satisfied. When this rule is found, the engine can conclude, or infer, the Then-clause (condition); this will result to adding, removing or modifying information (facts) to its working memory. The inference engine will iterate this process until it reaches to the goal (Wikipedia). Figure 2.2 displays the algorithm of forward chaining process.

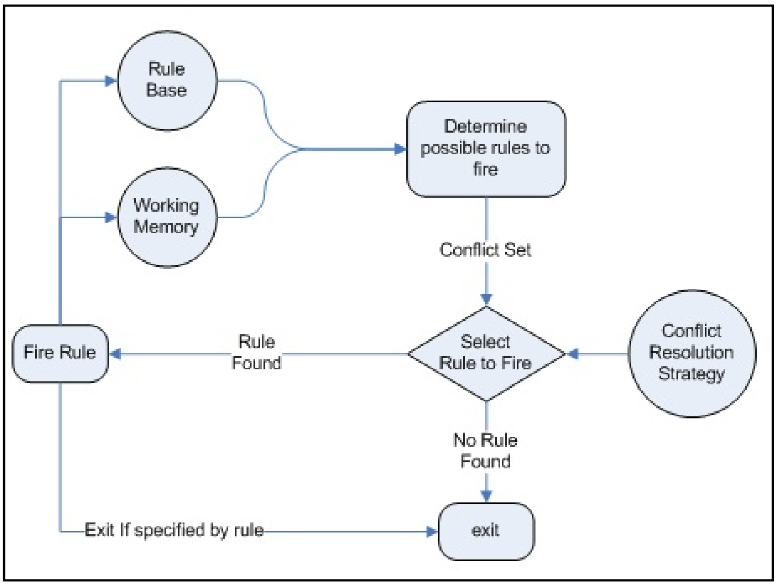


Figure 2.2: Forward chaining algorithm [7]

The rules are in the following form [9]:

left hand side (LHS) -> right hand side (RHS)

LHS is a collection of conditions which should be matched in the working memory, for the rule to be executed. The RHS contains the actions to be taken if the LHS conditions are met. The execution cycle is:

1) Select a rule whose left hand side conditions match the current state of the working memory.
2) Execute the right hand side of that rule. This may result in a change to the current state of the working memory (by adding, removing or modifying facts).
3) Repeat until there are no rules applicable.

Suppose that in a survey the goal is to conclude the monthly bonus of a person named "Jan", given the following knowledge base (a set of rules and facts). Assume these facts are available:

- Jan has age 62
- Pitter has age 59
- Jan has monthly-income > 12000
- Pitter has monthly-income < 12000

The rule base contains following rules:

- **If** X has age > 60 - **Then** X is a “Senior Citizen”
- **If** X has age <= 60 - **Then** X is a “Junior Citizen”
- **If** X is a “Senior Citizen” and monthly-income < 12000 - **Then** monthly-bonus = 2000
- **If** X is a “Senior Citizen” and monthly-income >= 12000 - **Then** monthly-bonus = 0
- **If** X is a “Junior Citizen” and monthly-income < 12000 - **Then** monthly-bonus = 1000
- **If** X is a “Junior Citizen” and monthly-income >= 12000 - **Then** monthly-bonus = 0

Now we want to perform forward chaining by following the pattern of an inference engine as it infer from the rules. With forward reasoning, the inference engine can derive that Jan is a “Senior Citizen”. This new fact is added to the working memory. Therefore we have the new fact:

- Jan is a “Senior Citizen”

Now having the new fact and the previous fact:

- Jan is a “Senior Citizen”
- Jan has monthly-income > 12000

The inference engine derives:

Jan is a “Senior Citizen” and monthly-income >= 12000

Then referring to the rule 4, the inference engine can derive:

monthly-bonus = 0

Therefore the monthly bonus of Jan is “0” and by iterating this process for Pitter, the rule 7 would be satisfied and we get the monthly–bonus of pitter = 1000. This process is shown in Figure 2.3 and we see that this is a bottom–up process. This is the method which the inference engine uses for reaching the answer. The inference engine starts from the data and uses rules to infer over those data and reach the answer, thus this method is called “data–driven”. We can also say that this method is a bottom-up method for the reason that it initiates from introductory data to reach to the final goal.

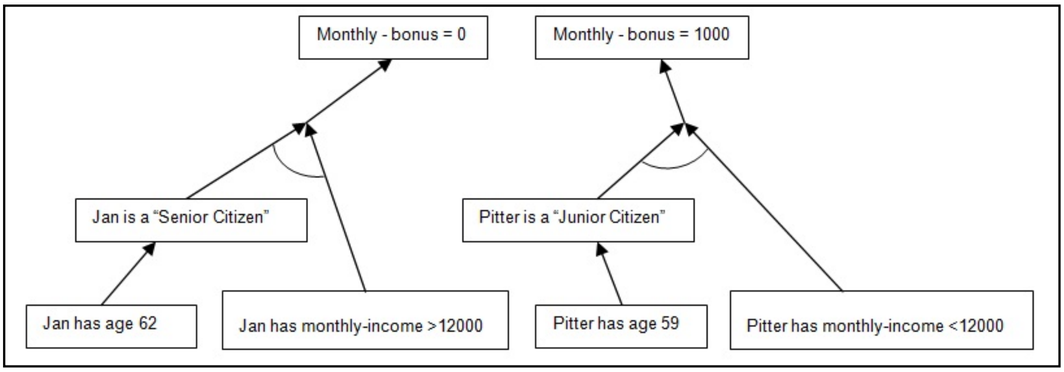

Monthly - bonus = 0
Monthly - bonus = 1000
Jan is a "Senior Citizen"
Pitter is a "Junior Citizen"
Jan has age 62
Jan has monthly-income >12000
Pitter has age 59
Pitter has monthly-income <12000

Figure 2.3: Forward chaining process

Backward chaining

Backward chaining is "goal-driven"; therefore its process is the opposite of the forward chaining process. For backward chaining we start with a goal which the inference engine tries to satisfy. If the main goal cannot be satisfied, then the engine searches for goals that it can satisfy; these are known as sub goals. Satisfying sub goals will help to reach the current goal. This process continues until the initial goal is confirmed. The following example clarifies the process of backward chaining.

Suppose that these data are given:

- U^W=> Y
- Z^Y=> X
- O=> W
- P=>W

The question or goal is X=?. Using backward chaining in order to reach the main goal we need the answer of two sub goals: Z and Y. There is no sub goal for Z, but in order to reach the second sub goal, Y, its sub goals, U and W should be satisfied. This procedure continued until no other sub goals remain. Preparing the answer for all the sub goals, will result in initial or main goal satisfaction. Figure 2.4 illustrates this process from left to right.

The logical representation for the result would be: X = Z. (U. (O+P)). This result is acquired step by step:

$X = Z . Y \rightarrow X = Z . (U . W) \rightarrow X = Z . (U . (O + P))$

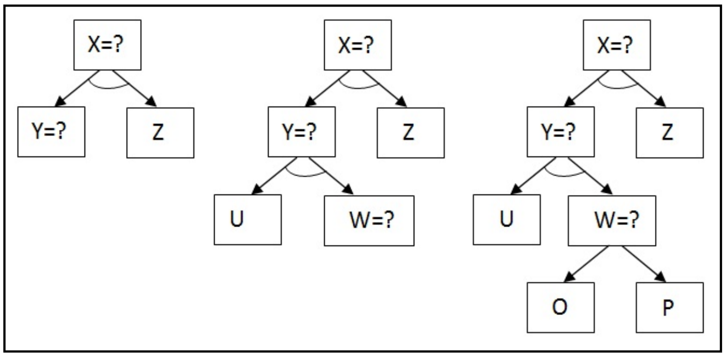


Figure 2.4: Backward chaining process

This process also can be explained with the previous example which we made for forward chaining. As before, we have the following rules:

- **If** X has age > 60 - **Then** X is a “Senior Citizen”
- **If** X has age <= 60 - **Then** X is a “Junior Citizen”
- **If** X is a “Senior Citizen” and monthly-income < 12000 - **Then** monthly-bonus = 2000
- **If** X is a “Senior Citizen” and monthly-income >= 12000 - **Then** monthly-bonus = 0
- **If** X is a “Junior Citizen” and monthly-income < 12000 - **Then** monthly-bonus = 1000
- **If** X is a “Junior Citizen” and monthly-income >= 12000 - **Then** monthly-bonus = 0

Using backward chaining we want to reach to the goal: “Citizens who has monthly-bonus = 1000”. We use the available data and try to satisfy sub goals in order to reach to the main goal. Available facts are:

- Jan has age 62
- Pitter has age 59
- Mike has age 28
- Jan has monthly-income >12000
- Pitter has monthly-income <12000
- Mike has monthly-income >12000

We start from the main goal: “monthly-bonus = 1000”. In order to satisfy this goal, we have two sub goals: “monthly-income < 12000 “and “is a “Junior Citizen””, that both of them should be satisfied. The sub goal: “is a “Junior Citizen””, has a sub goal itself: “has age <= 60”. Using the available data, for this sub goal: “has age <= 60”, we got the answer: Pitter and Mike (“Junior citizen”), while for the sub goal: “monthly-income <12000 “, we come to the answer: Pitter. Thus the result for the goal: “Citizens who has monthly-bonus = 1000”, would be: Pitter.

2.5 The Rete algorithm

The Rete algorithm is a pattern matching algorithm designed by Dr Charles L. Forgy of Carnegie Mellon University [8]. Rete is an efficient algorithm which enables matching facts against patterns described in rules. The algorithm can be used for systems containing anything from a few to many patterns and objects. The Rete algorithm uses concepts such as Rule set, Rules and Facts. A rule set is a knowledge base consisting of one or more rules. Every rule in the rule set represents some additional knowledge and is usually in the form of if → then. Here is a simple rule, where if, then and assert are keywords:

- **If** GMAT score >= 600 **Then assert** status = "passed"

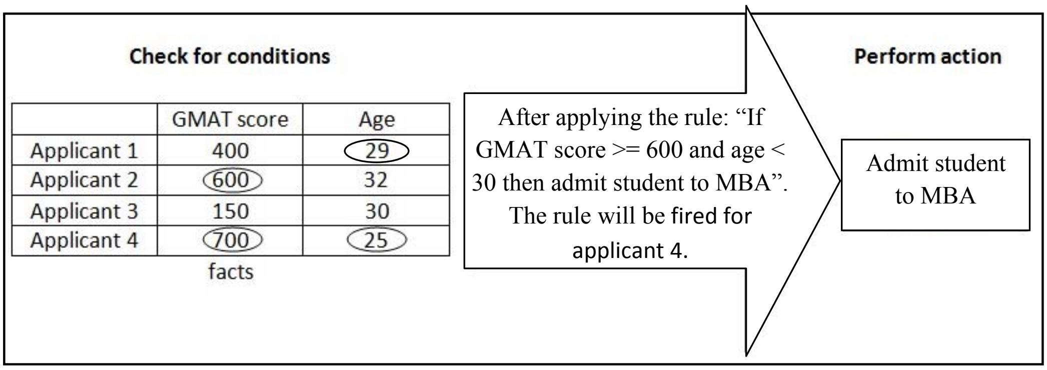
The If-part represents a condition and the Then-part represents an action. In a rule, we can have more than one condition and in that case the conditions should be joined by the logical operators: AND or OR. The Then-part also can contain one or more actions. The above rule example means that if the GMAT score of a person is equal to or larger than 600, then his status for the GMAT test is passed. In order to check someone's status, we need the person's score at the GMAT test and this data is called a fact. The following example represents a rule which has more than one condition: GMAT score >= 600 and age < 30.

If GMAT score >= 600 and age < 30 **Then Admit** student to MBA program

To check the above rule we need two data or facts: one for score and one for age. Therefore, in order to admit a student to the MBA program, his score should be equal or larger than 600 and his age should be less than 30. This is a simple rule and in our real life we may use many facts where the rules should manage them.

A complete rule-set should be given to the Rule Engine. The Rule Engine matches each rule in the rule set with given facts to decide whether to fire (execute) the rule or not. For example, considering a set of facts for some applicants for a MBA program, presented at the left side of Figure 2.5, after applying this rule" If GMAT score >= 600 and age < 30 Then Admit student to MBA program", the rule will be fired for applicant 4 and the action "Admit student to MBA program" will be performed for this applicant (displayed at the right side of Figure 2.5).

This process is called pattern matching. In each pattern matching process, the list of facts may be modified by adding or removing facts from the list. These changes may cause previously not fired patterns to be fired. During each cycle, the rules that are fired should be maintained and updated. If a Rule Engine checks each rule for all the facts although they are not modified, this will slow down the process. By maintaining what it has already matched from cycle to cycle and then computing only the changes for the added or removed facts, the process can be performed faster, which is done by the Rete algorithm [8].



	GMAT score	Age
Applicant 1	400	29
Applicant 2	600	32
Applicant 3	150	30
Applicant 4	700	25

After applying the rule: “If GMAT score >= 600 and age < 30 then admit student to MBA”. The rule will be fired for applicant 4.

Figure 2.5: Pattern matching process

The inference cycle for the Rete algorithm

In the Rete algorithm for applying each rule an inference cycle should be performed. Each inference cycle involves three main actions: match, select and execute. In the matching phase, the conditions of the rules are matched against the facts to see which rules are to be executed. The rules that their conditions satisfied are stored in a list called agenda for firing. From the list of rules, in the agenda, one of the rules is selected for execution by performing the actions on the right hand side of the rule. The action may be an assertion, execution etc. Figure 2.6 illustrates a high level view of rules, facts and agenda for performing pattern matching [8]. For more details about building of the Rete network refer to [10].

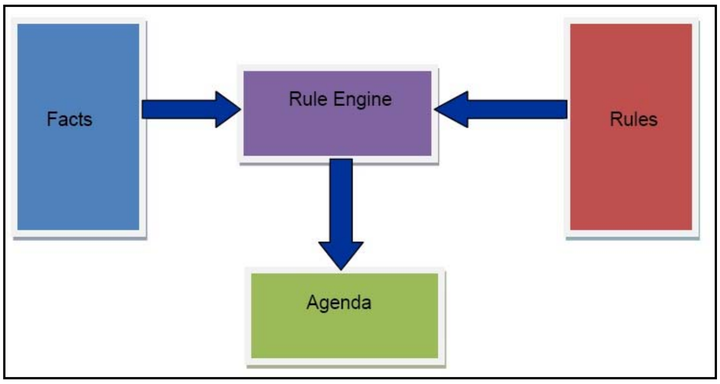


Figure 2.6: Pattern matching: rules, facts and agenda [8]

3 Methodology

The goal of this section is to give an overview of the reasoning method used in this project for developing an event chains generation system, and the reasons for selecting that method.

3.1 Reasoning methods

Forward and backward chaining methods are two main methods of reasoning. These two methods can be used in this project for reasoning over cause-consequence information represented as an ontology. The methods have been explained in Chapter 2. Considering the advantages and disadvantages of these methods, one of them should be selected for reaching to the expected goal (generating event chains automatically). The following information explains how these methods can be applied in this project and then the methods are compared for selecting the better choice.

Forward chaining implementation

In the forward chaining method the inference engine uses a bottom–up process for reaching to the answer. In this method the inference engine starts form the data and uses rules to infer over those data to reach a certain goal, and for this reason this method is called “data–driven”. To explain how this method can be applied in this project, consider the following example.

Assuming Information about direct connections between events, the cause-event leading to the consequence-event, for a selected system is available. Connecting these connections manually results in a set of event chains which have been illustrated in Figure 3.1. It is expected that the developed system will be able to generate some of these chains automatically. Desired chains should end with a specific goal. In this example we assume that the event “E” is the final goal. The result of implementing the forward chaining method on direct cause-consequence connections of Figure 3.1 is shown in Figure 3.2. As can be seen in Figure 3.2 the forward chaining process executed in seven phases over event connections to reach the answer (“E”).

Backward chaining implementation

Backward chaining is "goal-driven"; therefore its process is the opposite of the forward chaining process. This method starts from the main goal and, using a set of rules, goes back to reach an answer. Figure 3.3 illustrates implementing backward chaining over available events connections. Backward chaining is also performed during different phases (three phases) to reach the answer in this example.

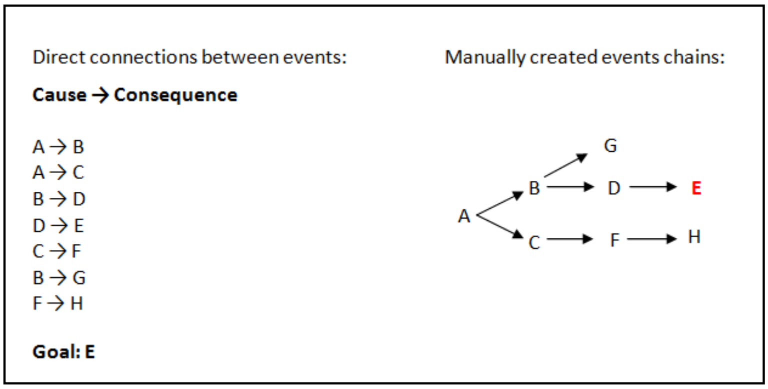

Direct connections between events:
Cause → Consequence
A → B
A → C
B → D
D → E
C → F
B → G
F → H
Goal: E
Manually created events chains:

Figure 3.1: Cause-consequence connections and the manually created event chains

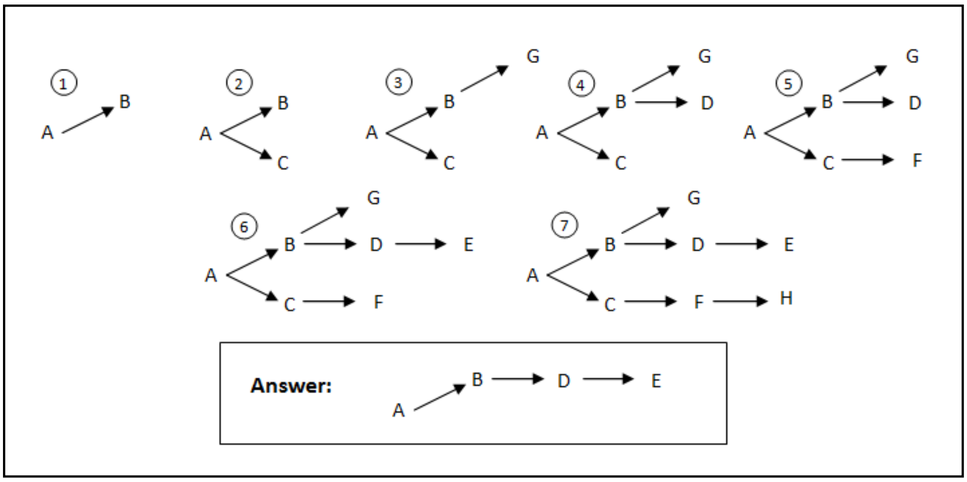


Figure 3.2: Forward chaining implementation

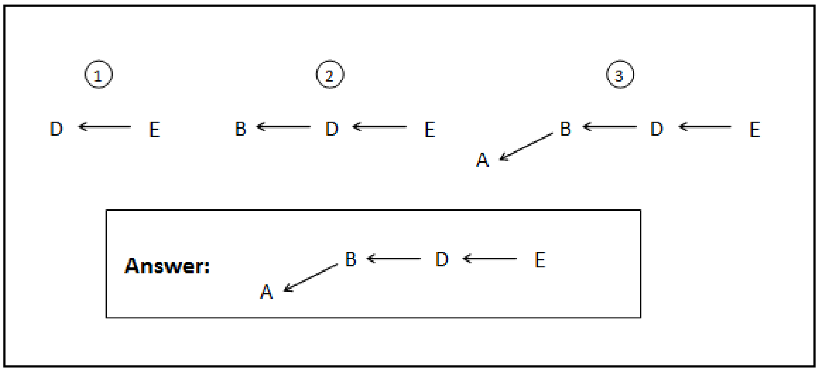


Figure 3.3: Backward chaining implementation

Forward chaining vs. backward chaining [11]

In this part, these two methods will be compared and the advantages and disadvantages of each method will be discussed.

Forward chaining advantages:

1) Forward-chaining is better than backward-chaining since by adding new data, new inferences will be made. The reason for this is that during the inference process, additional information is provided and, this will be used for new inferences. This is an important feature for dynamic situations in which conditions are likely to change.
2) Forward chaining works well when the problem starts with gathering information and resulting in final conclusion.
3) Forward chaining works well for specific type of problem solving such as monitoring, control and interpretation.

Forward chaining disadvantages:

One disadvantage of forward chaining is that the system traverses all possible rules, even though it only needs to go over a few rules to reach to the conclusion.

Backward chaining advantages:

1) Because backward chaining focuses on the given goal, so only the rules that are related to the goal are traversed and it only searches that part of the knowledge base which is relevant to the current problem.
2) Using one of the benefits of backward chaining is that the user doesn't have to explicitly write rules for the sub goals. The Rule Engine generates the sub goal for each object.
3) Backward chaining is useful for cases in which we have a hypothesis (final goal) and we want to check if it can be proven.
4) Backward chaining is a useful technique for solving problem such as diagnostics, prescription and debugging.

Backward chaining disadvantages:

Backward chaining implementation is complex.

3.2 Method selection

After comparing these two methods, we need to decide whether we should use a forward chaining method or a backward chaining method. The choice between these methods depends on what kinds of problems we are going to solve. Based on the above information, both methods can be used in this project for generating a set of event chains.

As described in the previous section, using forward chaining method is effective for dynamic situations in which conditions are likely to change. The reason for this is that during forward chaining all possible connections are created. Then by applying any changes to the conditions (for example modifying the final goal) desired chains can be generated using the previously created connections. One disadvantage of using forward chaining method is that, the system traverses all possible rules, even though it only needs to go over a few rules to reach the conclusion. This disadvantage is important for systems that have resource limits. The example in Section 3.1 demonstrates this feature of forward chaining. Using forward chaining we reached the answer after seven phases while, using backward chaining the answer is obtained after three phases.

In this project, sequential dependent events of a system (steam boiler) have been selected; only some of these sequential events end with an accident event (final goal). Like the example in Section 3.1, applying backward chaining can be useful in this project, since backward chaining focuses on the given goal (the accident event), thus only the rules that are related to the goal are traversed and the result will be provided faster; but the main argument against selecting this method is its implementation complexity. Since implementing forward chaining is simpler than implementing backward chaining and backward chaining is complex and more time consuming; the forward chaining method has been selected as a reasoning method in this project. Existing systems that supports forward chaining implementation will be discussed in the next chapter.

4 Existing systems for supporting “forward chaining” method

In the previous chapter we compared the forward and backward chaining methods and we decided that applying the forward chaining method would be the best choice for this project; The main reasones being that forward chaining can be implemented simpler and faster than backward chaininng. The forward chaining method should be implemented using an inference system. Due to time limitations in this project, selecting a system which enables us to implement a forward chaining method in 3-4 month (considering the time needed to learn how to use the system) is important. Some inference systems that support forward chaining method are Jess/JessTab, Algernon, OOPS and Drools. These inference systems are evaluated with respect to important characteristics that are required for this project. Finally Jess/JessTab and Algernon rule-based inference systems were selected. In this chapter these inference systems (Jess/JessTab and Algernon) will be explained and evaluated based on the required characteristics for this project and finally the more appropriate system is selected for implementing the forward chaining method.

4.1 Important characteristics

When an inference system shall be used for reasoning over a knowledge base, there are important characteristics that should be considered. The following characteristics are evaluated for the Jess/JessTab and Algernon inference systems in Section 4.2 and 4.3 and then based on these evaluation in Section 4.4 a choice is made.

Functionality

- The system supports forward chaining implementation.
- Using the system it is possible to call external functions.

User experience

- The system should be obtained and installed easily.
- The syntax should be readable.
- To learn how to use the system a complete documentation should be available.

Engineering

- In this project we use an ontology, developed using Protégé as a knowledge base. The system must thus be able to operate on ontologies.
- The selected system should be robust.

4.2 Algernon

Algernon is a rule-based inference system which is implemented in Java and interfaced with Protégé (a Protégé tab plugin). Algernon performs forward and backward reasoning over frame-based knowledge bases and efficiently stores and retrieves information from ontologies and knowledge bases [12]. Algernon syntax has two main parts: Paths and Clauses. A clause represents a relation, for example: (name "Jane"). A path is a sequence of clauses, for example: ((name "Jane")(family "Wei")).

System evaluation

Functionality

- Algernon supports forward chaining rules.
- Using Algernon it is possible to call Java functions for non - knowledge base calculations.
- It can be integrated with java.

User experience

- Algernon can be obtained from the link: (http://algernon-j.sourceforge.net/). Some problems occured during Algernon installation.
- It has an Interpretive scripting language.
- A good tutorial is available for the system.

Engineering

- Algernon has a concise way to retrieve information from a knowledge base.
- It has the ability to read and write Protégé knowledge bases and has straightforward access to ontology classes and instances.
- Algernon operates directly on the Protégé knowledge bases rather than requiring a mapping operation.
- This system operates directly on Protégé frames.
- Using Algernon it is possible to have access to multiple concurrent knowledge bases.

4.3 Jess/JessTab

4.3.1 Jess

Jess (Java Expert System Shell), is a Rule Engine for the Java platform. Using Jess, it is possible to build Java software that has the capacity to "reason" about given knowledge using a set of declarative rules. It provides a complete environment for the construction of rule and/or object based expert systems. Jess uses Common LISP (CLIPS) type syntax to describe rules and facts. To use it, we should specify logic in the form of rules using one of two formats: the Jess rule language or XML. In addition some data should be provided for the rules to operate on. Jess is one of the fastest Rule Engines available. Its powerful scripting language gives access to all of Java's APIs. Jess includes a full-featured development environment based on the Eclipse platform [13].

System evaluation

Functionality

- Jess supports forward chaining implementation.
- Jess uses the Rete algorithm to process rules, which improve the speed of forward chaining by limiting the effort required to re compute connections after a rule is fired.
- Jess can manipulate and reason about Java objects, and it is also a powerful Java scripting environment, which enables creating Java objects, call Java methods, and implement Java interfaces.

User experience

- Jess is available at no cost for academic use and can be installed easily.
- The Jess syntax is readable.
- All data is structured as lists and is therefore easy to understand.
- A complete documentation is available for learning the system.
- Easy to learn and use.

Engineering

- Using Jess, it is possible to operate on ontologies.

4.3.2 JessTab

JessTab has all the features mentioned for the Jess expert system, since JessTab is a tab plug-in for running Jess inside Protégé. It is a bridge between Protégé and Jess. JessTab is a translator between two formats: Protégé format and Jess format. It mirrors Jess definitions in Protégé knowledge bases and enables Jess programs to manipulate the Protégé knowledge bases. This is achieved by mapping the Protégé knowledge bases to Jess assertions [14]. These features can be added for the JessTab:

- Protégé – Jess integration, JessTab provides Rule-based reasoning in Protégé.
- There is no support for JessTab plug-in in version 4 of Protégé; it is supported in version 3.

4.4 Conclusion on assessment

After explaining the Jess/JessTab and Algernon inference systems and evaluating them based on the required characteristics for this project, an appropriate system can be selected. Using the selected system, it should be possible to implement the forward chaining method in the limited time of this project. In this period, the knowledge for using the system should be acquired. Based on the above evaluation of the systems (Jess/JessTab and Algernon) and the following information:

- Algernon syntax is readable, but defining forward chaining rules which can be applied over cause-consequence concepts is more understandable using Jess language.
- Some problems during Algernon installation occurred, but the Jess is installed easily.
- Developing the Jess rules in the Eclipse platform makes the interaction between java codes and Jess rules easier than using the JessTab and developing rules in Protégé. It is therefore more efficient to use the Jess than JessTab.

It can be concluded that, implementing forward chaining using the Jess inference system is the better choice in this project.

5 Developing an event chains generation system

In this chapter we explain how we can generate a set of hazardous event chains for a system automatically, using the information of the system stored as an ontology. For this purpose, a system is developed and the developing process is described in full details. In addition, the developed system is tested with a real world example (a simplified steam boiler) to explain how the system works using an ontology (steam boiler ontology) to generate a set of event chains. Automatically identifying event chains can facilitate safety analysis for a complex system. Identifying these event chains that end with an accident can help us to prevent them from occurring.

5.1 Description of the system

5.1.1 Introduction

The main focus of this project is facilitation of safety analysis by reasoning about causes and consequences concepts of a system. The system developed in this project enables generating a set of event chains for a specific domain using a pre-defined ontology for that domain. Finally, the developed system will be tested with a real-world example.

Domain experts possess great knowledge about a specific domain, the domain rules and its processes. They can express the logic of the domain in their own terms and know the relations between the terms. Using the knowledge of a domain expert for a specific domain would be useful for safety analysis. Domain experts know the facts in their domains (what can cause a failure and the consequence of a specific failure to the system). Using this knowledge of a domain and together with experiences for a domain similar to this domain provides the safety analyst with information that can be used efficiently for identifying possible hazardous events for this system. But the main challenge is that how this knowledge can be manipulated.

Reasoning is a method for improving and manipulating knowledge of a specific domain. In this project we use a reasoner to improve and manage the domain knowledge of a system. The reasoner enables generating a set of event chains by manipulating the knowledge of a system that has been stored as an ontology. Since the main focus of this project is safety analysis, the safety knowledge of the domain (knowledge about causes and consequences concepts) is the main part which is used for reasoning.

All potentially dangerous events, the events' consequences and the events' cause need to be identified for safety analysis. The sequence of these dependent events or the event chains describes how the effect of a specific failure is the cause of another failure. Each event chain starts with an initial event and ends with a final event. We assume that the final event is an unwanted event that occurs in the system's environment, for instance to a building or to humans. It can be assumed that the final event is the unwanted accident. Thus, in the system which is developed in this project, each event chain starts with an initial event and ends with a final event which may be an accident.

5.1.2 System perspective

This section describes the developed system (event chains generation system) and its development process. Figure 5.1, gives an overview of the event chains generation system.

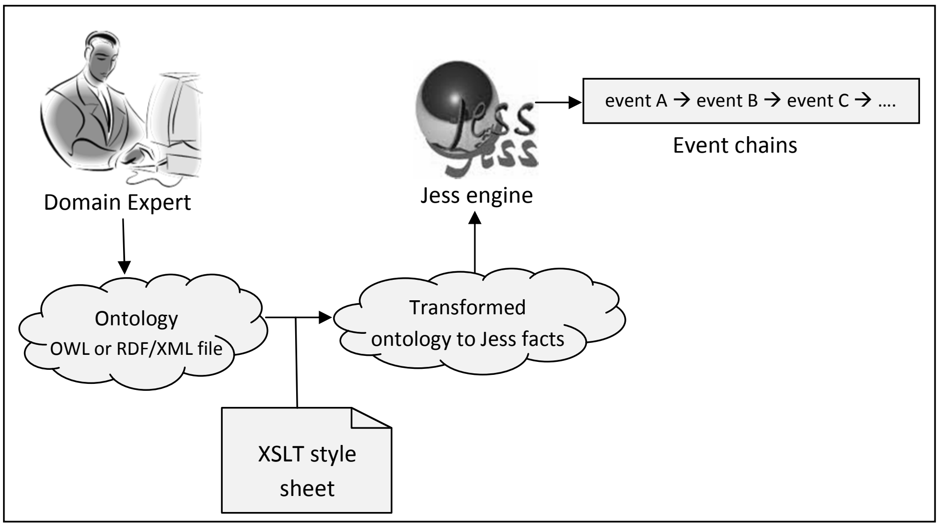


Figure 5.1: Event chains generation system overview

In this project the following four main actions need to be performed.

1) **Creating an ontology**

   This task should be done by the domain expert. The domain expert specifies the concepts of a domain and relationships that exists between these concepts in a formal way. An ontology enables us to perform more actions over knowledge of a specific domain. In addition, an ontology defines a common vocabulary for people who need to share information in a domain and enables reusability of information. The developed ontology is used as a knowledge base, a mean for storing our data (facts) of a specific domain, which further reasoning will be performed over them. However, because the main focus of events chain generation system is facilitation of safety analysis, the concepts which are related to the accident reports and hazardous events should be contained in the ontology. These concepts should be provided by a safety expert. For this purpose, in Section 5.3, creating an ontology which contains event cause-consequence concepts will be described. In order to perform forward chaining over the ontology concepts, the cause-consequence concepts of events should be defined in a specific way which is explained in Section 5.3. In this project, the Protégé tool has been used for developing an ontology and the created ontology can be saved as an OWL or RDF/XML file.

2) **Ontology transformation to the Jess facts**

Using Protégé we can develop an ontology and a reasoner can find facts that are implicit in the given ontology and then makes further actions over those facts. Jess is an inference engine and will be described later. We use the Jess inference engine for reasoning about domain knowledge which are stored as an ontology. But the Jess engine cannot infer the knowledge in the format of an ontology (for instance an OWL file). The Jess inference engine uses .clp files. Therefore the ontology must be translated to the Jess format, and the Jess engine can then be used for reasoning. This transformation is done using XSLT style sheets [15]. The transformation process will be explained in Section 5.5.

3) **Extracting hazardous cause-consequence facts from the transformed ontology**

The transformed ontology to the Jess facts contains all the ontology information, but we need only the information about events. Facts that give information about causes and consequences concepts must be extracted from other facts. The extracted cause-consequence facts can be used for forward chaining. Details of this process are described in Section 5.6. A related task have been done in [16], where the system architecture (components and connections), internal states of the components and state transitions are extracted and modeled using the results of text mining which performed in [17].

4) **Forward chaining**

Based on the discussions in Chapter 3, the forward chaining method is selected as a reasoning method in this project for reasoning over cause-consequence concepts. The main reason for this selection was that forward chaining implementation is simpler than backward chaining. Although backward chaining method has some advantages which can result in faster performance of the system, but its implementation is complex and forward chaining method is the better choice for this project. The transformed ontology, from which we extract cause-consequence facts, now is ready to be used by the Jess engine to perform forward chaining. Forward chaining is implemented by defining a set of rules and applying these rules to the Jess facts. In this project, Jess rules has written in the Jess rule language and developed on the Eclipse platform. Details of forward chaining implementation will be described in Section 5.7. Implementing forward chaining over Jess facts result in generating a set of chains.

5.1.3 System features

- **The system can extract event cause-consequence concepts (event instances and their “CanCause” relationships) form the ontology.** The system extracts event concepts from the ontology if the ontology is developed based on the specified format - the ontology which contains an Event class, event instances and CanCause property. Then the forward chaining can be performed over the extracted events.

- **The system can generate a set of event chains based on the inserted initial event.** In Eclipse, when the run button is pressed the system performs forward chaining over the Jess facts in the working memory. This forward chaining is performed based on the defined rules. After pressing the run, the user must insert an event as the initial event. The Jess engine then traverses the facts in the working memory based on the rules. When this process is complete, the event chains that start with the inserted initial event and ends with an accident will be displayed.

- **In case of ontology modification, the event chains can be updated automatically.** When the ontology is modified by adding or removing events or event relationships, event chains must be updated based on the ontology modification. This can be done easily in this system. After saving the modified ontology, the only task is to transform XML syntax of the OWL file to the Jess assertions. Then the forward chaining process will be performed automatically over the modified ontology. This may result in generating new chains or modification to the previously generated chains. Performing this task (modifying an ontology and updating event chains) manually for a large system with various events and events relationships is difficult.

- **Multiple ontologies can be used simultaneously for reasoning.** Related chains will be generated based on the inserted ontologies. This feature would be useful when we require to use more than one ontology simultaneously for safety analysis, for example a domain ontology and an environment ontology.

5.1.4 System assumptions

To develop the event chains generation system the following assumptions have been made:

- **An ontology which contains cause–consequence concepts**
  The system performs forward chaining over the events based on a given ontology. This can be performed if the ontology contains event cause-consequence concepts provided. These event concepts are developed based on a specific format; the ontology which contains an Event class, event instances and CanCause property. For example, to define the following concept in the ontology:

  “event A **can cause** event B” or “event A $\xrightarrow{\text{CanCaus}}$ event B”

  First we create a new “Event” class, and then we add the two events (event A and event B) as instances of this class. For making a relationship between these two event instances we define a CanCause property.

- **The most harmful event to the environment is named an accident in the ontology**
  The system generated events chain based on the inserted Initial event and final event that is an accident. Therefore, in the ontology, the most harmful event that occurs in the system’s environment and causes an unwanted effect is named an accident in the

ontology. Therefore an events chain which starts with the inserted initial event (event A) and ends with the final event (accident) would be like the following:

Intiltal event-> event B-> event C-> event D-> accident

5.1.5 System Constraints

Currently the system can print event chains with limited number of events in a chain. The rules for creating all connections in a chain are already defined; therefore we only require defining a rule which enables printing unlimited number of events in a chain. This rule can use the event connections created form other defined rules, for printing.

Also, when the system is tested with some examples, we identified if the cause-consequence concepts in the ontology contains connections between events, like:

event A **can cause** event B, event A $\xrightarrow{\text{CanCaus}}$ event B
event B **can cause** event A, event B $\xrightarrow{\text{CanCaus}}$ event A

This will result in generating incorrect chains with the event chains generation system. For example, the event chain presented in Figure 5.2 cannot be generated by the system, since there is an inverse connection between event B and event C.

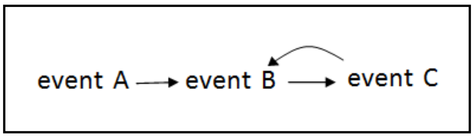


Figure 5.2: Example of a chain that cannot be generated by the system

5.2 General requirements

In order to develop an event chains generation system there are some requirements that should be available to the user. This section describes these requirements, and explains how they are essential for this system.

5.2.1 Protégé

For developing ontologies in the OWL format, the Protégé tool has been used as an ontology development tool. Protégé is a free, open-source platform that provides constructing domain models and knowledge-based applications with ontologies. Protégé can be customized to provide domain-friendly support for creating knowledge models and entering data. An ontology describes the concepts and relationships that are important in a particular domain, providing a vocabulary for that domain as well as a computerized specification of the meaning of terms used in the vocabulary. In recent years, ontologies have been adopted in many business and scientific communities as a way to share, reuse and process domain knowledge. Ontologies are now central to many applications such as scientific knowledge portals, information management and integration systems, electronic commerce, and semantic web services [5].

Using Protégé OWL ontologies can be developed. For describing the process of developing an ontology using Protégé, we wrote a manual which can be found in [18]. A reasoner can find facts that are implicit in the developed ontology and then makes further actions over those facts. An ontology enables us to specify concepts of a domain in a machine-interpretable format. An OWL ontology may include descriptions of classes, properties and their instances.

For this project the version 4 of the Protégé software has been used and the download link is: http://Protégé .stanford.edu/download/registered.html#p4.3

5.2.2 Jess

Jess is a Rule Engine for the Java platform, more explanation about this Rule Engine is provided in Section 4.3. To use Jess, we should specify logic in the form of rules using one of two formats: the Jess rule language or XML. In this project, the Jess rule language has been used for logic specification. Also some data should be provided for the rules to operate on. When we run the Rule Engine, rules are applied to the provided data and may result in creating new data. More details about how the Jess engine works will be described in Section 5.4 .Jess language codes can be developed in any text editor, but Jess comes with a full featured development environment based on the Eclipse platform. Jess uses the Rete algorithm to process rules. This algorithm was described in Chapter 2. For using Jess Rule Engine, there are some requirements which are specified in Appendix A.

Jess licensing

Jess can be licensed for commercial use, and is available at no cost for academic use (licensing terms can be asked from Craig Smith at casmith@sandia.gov). A trial download is also available [19].

The Jess Developer's Environment

Jess 7 includes an Eclipse-based development environment. There is an editor, a debugger, and a Rete network viewer. The Jess Developer's Environment can be installed using the instructions specified in: http://www.jessrules.com/jess/docs/71/eclipse.html#.

5.2.3 XSLT style sheets

Since we use the Jess inference engine for reasoning and the Jess engine cannot use knowledge in the form of an ontology (OWL file) thus, the ontology must be translated to the Jess format (.clp file), so that the Jess engine can reason about it. This transformation from OWL file to .clp file is done using XSLT style sheets. Details about how this transformation can be done will be provided in Section 5.5.

XSL stands for "Extensible Stylesheet Language" and XSLT stands for XSL Transformations. XSLT style sheets are used for transforming XML documents into other type of documents. In this project we use XSLT style sheets to transform the XML syntax of an OWL file to the Jess assertions, and the Jess engine is then able to reason about transformed ontologies and annotations. It should be noted that two XSLT style sheets may required:

1) An XSLT style sheet that transforms a file with OWL schema (in XML syntax) into a set of Jess assertions based on the OWL meta-model. The resulting assertions can be loaded into the Jess engine.
2) An XSLT style sheet that transforms a file with OWL annotations (in XML syntax) into a set of Jess assertions based on the OWL meta-model.

The resulting assertions can then be loaded into the Jess engine. For performing this transformation a file describing the OWL meta-model in the Jess language is also required. It is to be loaded directly into the Jess engine [15]. These two XSLT style sheets and the OWL meta-model file can be downloaded from: http://mcom.cs.cmu.edu/OWL/OWLEngine.html.

5.3 Designing an ontology

The ontology that we want to use as a knowledge base in the event chains generation system should contain possible accidents and causes and consequences concepts of the selected domain. Using Protégé we can develop OWL ontology and a reasoner can find facts that are implicit in the developed ontology. An OWL ontology may include descriptions of classes, properties and their instances. As we are going to use the information stored as an ontology for reasoning using the Jess inference engine, it is required that this ontology contains the following components:

an Event class - a CanCause property - event instances

Figure 5.3 shows each instance of the class Event has a CanCause relationship to itself. For example, event A has a CanCause relationship to event D: **event A CanCause event D.** An Event class that has the CanCause relation to itself is presented in Protégé in Figure 5.4.

All possible events in the domain should be defined as instances of the class Event. These event instances are related to each other with the “CanCause” property. Because each event instance can only has a relationship with another event instance, thus, the domain and range of the class Event should be itself, the class Event (Figure 5.5).

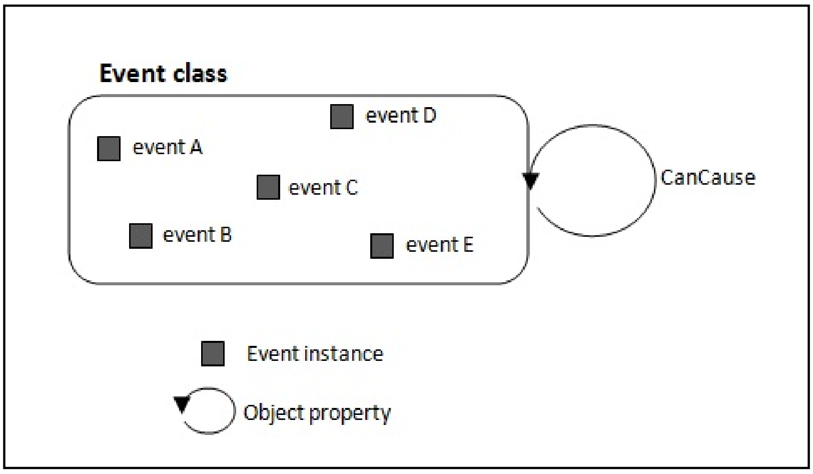
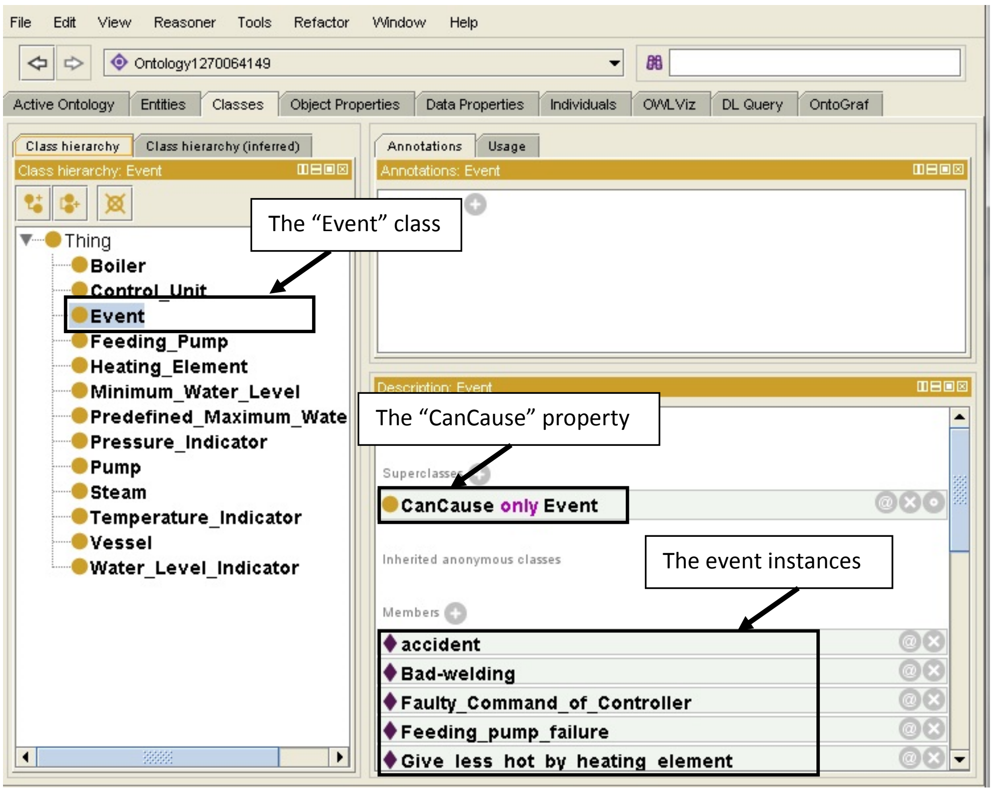


Figure 5.3: Illustration for ontology components which includes event concepts



Thing
- Boiler
- Control_Unit
- Event
- Feeding_Pump
- Heating_Element
- Minimum_Water_Level
- Predefined_Maximum_Wate
- Pressure_Indicator
- Pump
- Steam
- Temperature_Indicator
- Vessel
- Water_Level_Indicator

Superclasses: CanCause only Event

Members:
- accident
- Bad-welding
- Faulty_Command_of_Controller
- Feeding_pump_failure
- Give_less_hot_by_heating_element

Figure 5.4: Illustration of CanCause relationship of Event class to itself

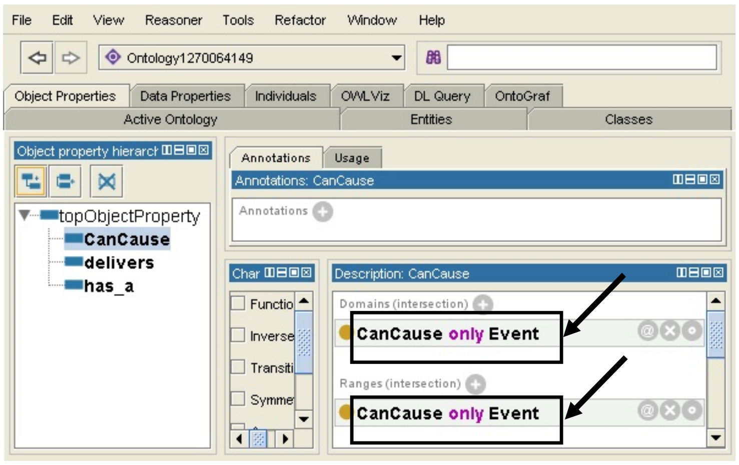


Figure 5.5: Illustration of CanCause property, its domain and range

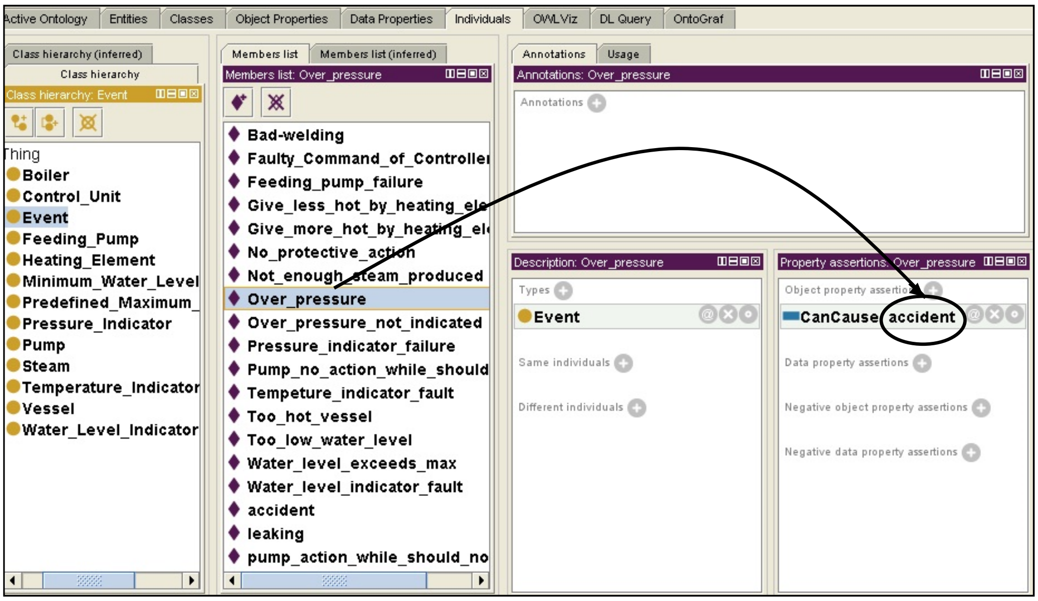


Figure 5.6: Steam boiler ontology which contains cause-consequence concepts

Figure 5.6 shows an example for the steam boiler ontology in Protégé (the steam boiler system will be described later) with the cause-consequence concepts added. First the Event class is defined in the ontology. Then a set of individuals are added to this class. Finally the CanCause property is defined as an object property. Based on the available cause-consequence data of the domain, each event instance relates to another event instance via a CanCause property. For example, considering the two event instances: “Over_pressure” and “accident”, these two events relate to each other via the CanCause property:

Over_pressure CanCause accident

This explains that the “Over_pressure” event can cause an “accident” event in the given domain.

5.4 How does Jess work?

Jess developed at Sandia National Laboratories by Dr. Ernest J. Friedman-Hill for creating rule-based expert systems. Rule-Based Expert Systems contains a rule base, a working memory (fact base) and an inference engine (Rule Engine). An inference engine has a pattern matcher, which should decide what rules to fire and when. In an inference engine an agenda schedules the order in which activated rules will fire and execution engine is responsible for firing rules and executing functions.

The inference Process

Figure 5.7 illustrates an overview of the inference process. The working memory contains a list of facts, which have previously been defined or will be asserted during the inference process. An inference engine matches the facts against the rules. Rules are in the following form:

LHS -> RHS

If LHS then RHS

If any fact in the working memory matches the LHS of a rule then the rule is activated and inserted to the agenda. Rules may be deactivated and removed from the agenda, if the matching facts are removed from working memory. Next step is rule execution or rule firing. A rule fires, if Jess executes the actions of RHS. This action may be a fact assertion/removal/modification to the working memory or it can be any function call. The sequence of actions is determined by agenda [9].

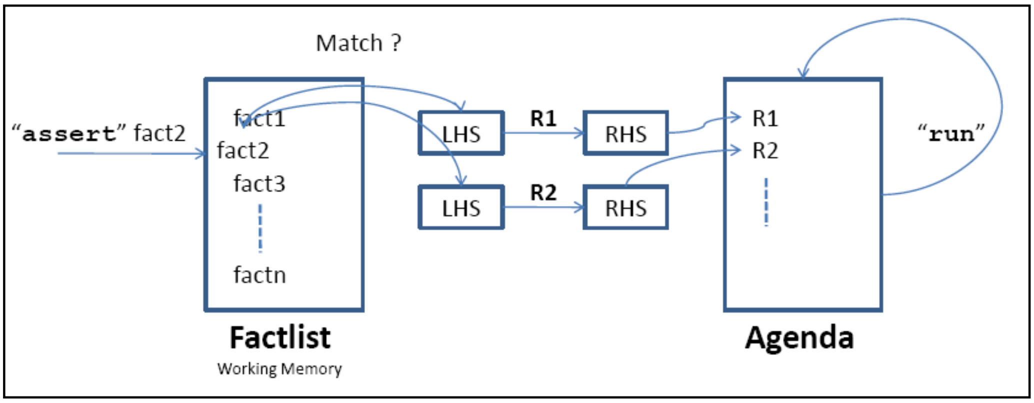


Figure 5.7: Inference Process [9]

Jess uses the Rete algorithm (described in Chapter 2) to match patterns.

Template, Fact and Rule in Jess

In this section we describe how templates, facts and rules should be defined in Jess. Jess manages a list of known facts. The following defines a Jess fact sample which gives some information about an event:

```
(deffacts    my_facts
        (my_event
                (event_name    "Over_pressure")
                (event_status   "active")))
```

Each fact should correspond to a template. A template is defined as following:

```
(deftemplate    my_event    "A sample deftemplate"
        (slot    event_name)
        (slot    event_status))
```

Assume that many hazardous events may occur in a steam boiler system. Using Jess rules we need to find special event facts and as response some actions should be performed. For example to explain the following rule:

If LHS Then RHS

IF the event “Over_pressure” is activated **THEN** the response is to open the safety valve

First, the relevant templates should be defined:

```
(deftemplate    my_event   "A sample deftemplate"
        (slot    event_name)
        (slot    event_status))
```

and

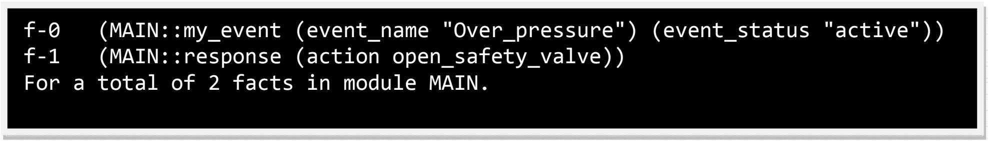
```
(deftemplate    response   "response template"
        (slot action))
```

then, defining the rule:

LHS -> RHS

```
(defrule    safety_action   "A sample rule"
        (my_event
                (event_name    "Over_pressure")
                (event_status   "active"))
=>
 (assert (response
                        (action  open_safety_valve))))
```

Thus, the Jess engine matches the facts against the rule, the rule is activated because the fact (LHS of the rule) exists in the working memory and after firing, and the action on the RHS is executed. The result of the execution would be an assertion of a new fact: (response (action open_safety_valve)) to the working memory. Therefore, after execution, there are two facts in the working memory and these facts are displayed in the console view in Eclipse. Figure 5.8 shows the displayed facts in the console view of Eclipse.



```
f-0   (MAIN::my_event (event_name "Over_pressure") (event_status "active"))
f-1   (MAIN::response (action open_safety_valve))
For a total of 2 facts in module MAIN.
```

Figure 5.8: Presentation of facts in the console view in Eclipse

5.5 Transforming an OWL file (in XML syntax) to a set of Jess assertions [15]

As described in Section 5.2.3, in this project we use XSLT style sheets to transform the XML syntax of an OWL file to the Jess assertions. The Jess engine is then able to reason about transformed ontology and annotations. To perform this transformation, three main files are required, two XSLT style sheets and an OWL meta-model in the Jess language:

1) Ontology Style sheet (OWLOntology2Jess.xsl): An XSLT style sheet that transforms a file with OWL schema (in XML syntax) into a set of Jess assertions.
2) Annotation Style sheet (OWLAnnotation2Jess.xsl): An XSLT style sheet that transforms a file with OWL annotations (in XML syntax) into a set of Jess assertions.
3) OWL Meta-model (OWL.clp): A file describing the OWL meta-model in the Jess language. It should be loaded directly into the Jess engine.

Figure 5.9 shows ontology and ontology annotation transformation to the CLIPS format (which is used by the Jess engine) via these three files.

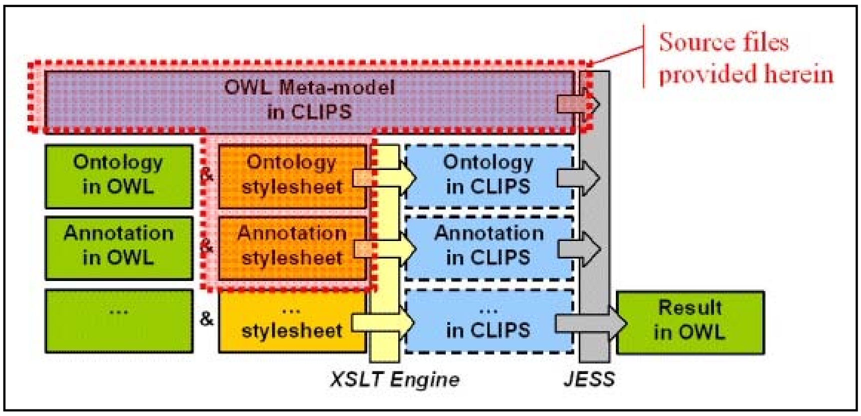


Figure 5.9: Ontology transformation over view [15]

In the ontology events are defined as following:

cause → consequence
event A → event B
event A **CanCause** event B

This explains that event A can cause event B or the cause for event B is event A or the consequence of event A is event B. After transforming the ontology, the ontology is converted to the Predicate-Subject-Object (PSO) triples. All events concepts (causes and consequences) are also transformed to the triple format. These triples can be asserted to the working memory and the Jess engine can infer about them. A template with three slots, Predicate, Subject and Object can be defined like this:

```
(deftemplate    triple "Template representing a triple"
        (slot predicate (default ""))
        (slot subject   (default ""))
        (slot object    (default "")))
```

It can be used for explaining:

Subject → Object or Subject -------------Predicate ------------> Object

This will be used further for mapping the causes and consequences concepts to PSO triples. Each slot should be filled with a string, this includes the namespace declarations. Namespaces provide a method to avoid element name conflicts. Namespace declarations are used as precise indications of what specific vocabularies are being used. A standard initial component of an ontology includes a set of XML namespace declarations enclosed in an opening rdf:RDF tag. These provide a means to unambiguously interpret identifiers and make the rest of the ontology presentation much more readable. A sample of this triple is shown below:

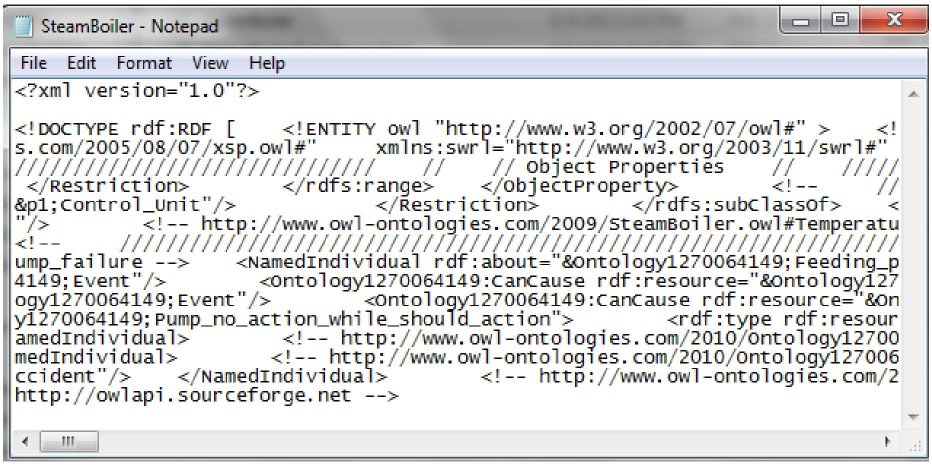
```
(triple
        (predicate "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#CanCause")
        (subject"http://www.owl-ontologies.com/2010/Ontology1270064149.owl#Over_pressure")
        (object "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#accident"))
```

The above triple represents the following concepts in the ontology:

Over_pressure---------- CanCause-----------> accident

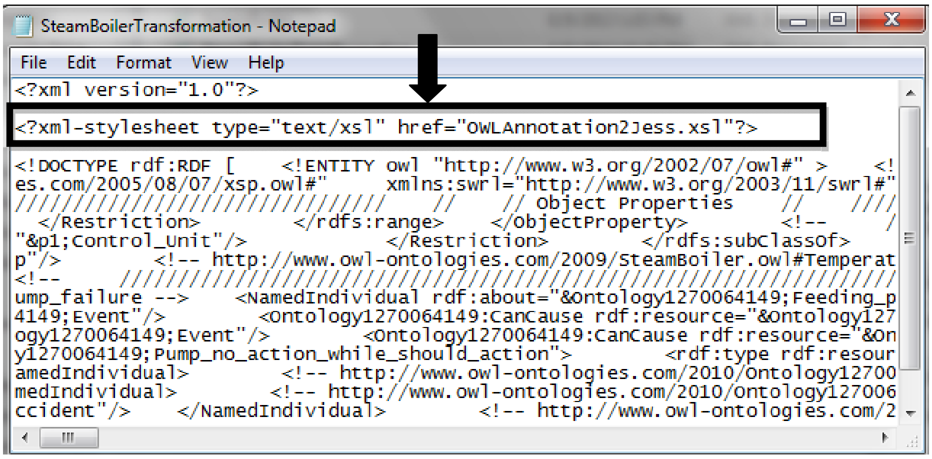
After transformation, these concepts can be asserted to the working memory and will be ready for forward chaining. In this project we need to transform the ontology annotation to Jess assertions. Since the event instances and their relationships in the ontology should be used for forward chaining, the “Ontology Style sheet” is not required and only the “Annotation Style sheet” is used. Other ontology information is not required, so we do not transform them. The transformation process is described in this section.

First of all, the OWL Meta-model (OWL.clp) file is loaded directly to the Jess. Next step is using the XML file of the selected ontology (Figure 5.10) for adding a new tag to it. This new tag “<?xml-stylesheet type="text/xsl" href="OWLAnnotation2Jess.xsl"?> ” should be inserted to the XML file(Figure 5.11).



```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >    <!
s.com/2005/08/07/xsp.owl#"     xmlns:swrl="http://www.w3.org/2003/11/swrl#"
///////////////////////////////    //    // Object Properties    //    /////
 </Restriction>      </rdfs:range>    </ObjectProperty>      <!--      //
&p1;Control_Unit"/>          </Restriction>       </rdfs:subClassOf>     <
"/>       <!-- http://www.owl-ontologies.com/2009/SteamBoiler.owl#Temperatu
<!--     ////////////////////////////////////////////////////////////////////
ump_failure -->    <NamedIndividual rdf:about="&Ontology1270064149;Feeding_p
4149;Event"/>       <Ontology1270064149:CanCause rdf:resource="&Ontology127
ogy1270064149;Event"/>       <Ontology1270064149:CanCause rdf:resource="&On
y1270064149;Pump_no_action_while_should_action">       <rdf:type rdf:resour
amedIndividual>      <!-- http://www.owl-ontologies.com/2010/Ontology12700
medIndividual>      <!-- http://www.owl-ontologies.com/2010/Ontology127006
ccident"/>    </NamedIndividual>       <!-- http://www.owl-ontologies.com/2
http://owlapi.sourceforge.net -->
```

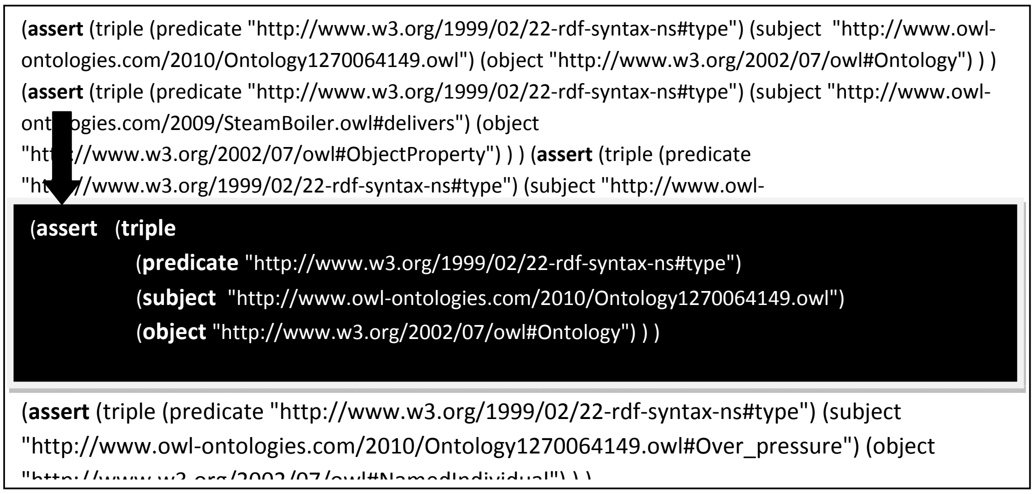
Figure 5.10: XML file of selected ontology


SteamBoilerTransformation - Notepad
File Edit Format View Help

```
<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="OWLAnnotation2Jess.xsl"?>

<!DOCTYPE rdf:RDF [    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >    <!
es.com/2005/08/07/xsp.owl#"     xmlns:swrl="http://www.w3.org/2003/11/swrl#"
///////////////////////////////    //    // Object Properties    //    ////
  </Restriction>      </rdfs:range>    </ObjectProperty>      <!--      /
"&p1;Control_Unit"/>          </Restriction>       </rdfs:subClassOf>
p"/>       <!-- http://www.owl-ontologies.com/2009/SteamBoiler.owl#Temperat
<!--     ///////////////////////////////////////////////////////////////////
ump_failure -->    <NamedIndividual rdf:about="&Ontology1270064149;Feeding_p
4149;Event"/>       <Ontology1270064149:CanCause rdf:resource="&Ontology127
ogy1270064149;Event"/>       <Ontology1270064149:CanCause rdf:resource="&On
y1270064149;Pump_no_action_while_should_action">       <rdf:type rdf:resour
amedIndividual>      <!-- http://www.owl-ontologies.com/2010/Ontology12700
medIndividual>      <!-- http://www.owl-ontologies.com/2010/Ontology127006
ccident"/>    </NamedIndividual>       <!-- http://www.owl-ontologies.com/2
```

Figure 5.11: Adding a new tag to the XML file of selected ontology

After adding the new tag, this file should be saved as a new XML file (for example Transformation.xml). The new XML file transforms the ontology annotations to Jess assertions. When the new XML file is executed, the result would be a set of facts that can be loaded to the Jess and the inference engine can use them for inference. Figure 5.12 shows an example of ontology which is transformed to some Jess assertions. It is important that, all three files, the Annotation Style sheet (OWLAnnotation2Jess.xsl), the ontology XML file and the XML file which a new tag added to it, are stored in the same folder.



```
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type") (subject  "http://www.owl-
ontologies.com/2010/Ontology1270064149.owl") (object "http://www.w3.org/2002/07/owl#Ontology") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type") (subject "http://www.owl-
ont  ogies.com/2009/SteamBoiler.owl#delivers") (object
"ht  ://www.w3.org/2002/07/owl#ObjectProperty") ) ) (assert (triple (predicate
"ht  //www.w3.org/1999/02/22-rdf-syntax-ns#type") (subject "http://www.owl-

(assert   (triple
              (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
              (subject  "http://www.owl-ontologies.com/2010/Ontology1270064149.owl")
              (object "http://www.w3.org/2002/07/owl#Ontology") ) )

(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type") (subject
"http://www.owl-ontologies.com/2010/Ontology1270064149.owl#Over_pressure") (object
"http://www.w3.org/2002/07/owl#NamedIndividual") ) )
```

Figure 5.12: Jess assertions after transforming the ontology annotations

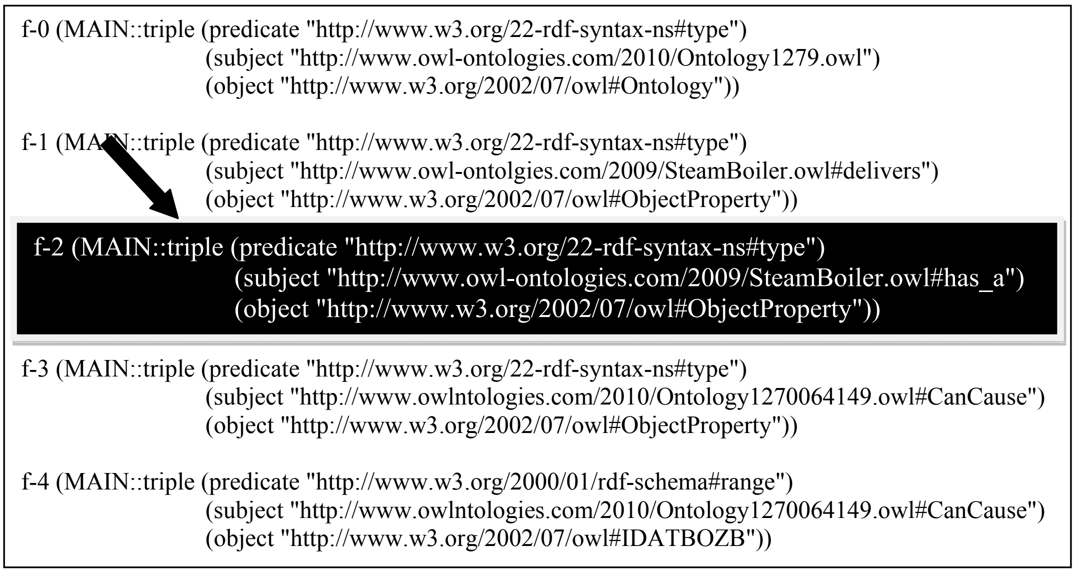
This transformation contains many Jess assertions, but we need only the triples that give information about causes and consequences concepts to perform forward chaining over them. Thus, these triples should be extracted from other Jess triples.

5.6 System design

This chapter explains the design of the complete system, including an overview of developed Jess files and their interactions. As described, the Jess inference engine is used in this project to manipulate and improve our knowledge of a specific domain. After building the ontology, the XML syntax of it should be transformed to the Jess assertions and the result of this transformation would be a set of Jess assertions which the inference engine can use them for inference. When we are sure that the Jess Developer's Environment installed correctly in Eclipse, we can start developing the events chain generation system. The event chains generation system contains three files:

OntologyTransformation.clp

This file contains the OWL metamodel which defines all the required templates and rules that used in the transformed OWL file. In addition, the transformed ontology annotations (Jess assertions) should be inserted into the specified part in the code. The result of executing this file would be a set of triple facts. Some of these facts are displayed in Figure 5.13.



```
f-0 (MAIN::triple (predicate "http://www.w3.org/22-rdf-syntax-ns#type")
                  (subject "http://www.owl-ontologies.com/2010/Ontology1279.owl")
                  (object "http://www.w3.org/2002/07/owl#Ontology"))

f-1 (MAIN::triple (predicate "http://www.w3.org/22-rdf-syntax-ns#type")
                  (subject "http://www.owl-ontolgies.com/2009/SteamBoiler.owl#delivers")
                  (object "http://www.w3.org/2002/07/owl#ObjectProperty"))

f-2 (MAIN::triple (predicate "http://www.w3.org/22-rdf-syntax-ns#type")
                  (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#has_a")
                  (object "http://www.w3.org/2002/07/owl#ObjectProperty"))

f-3 (MAIN::triple (predicate "http://www.w3.org/22-rdf-syntax-ns#type")
                  (subject "http://www.owlntologies.com/2010/Ontology1270064149.owl#CanCause")
                  (object "http://www.w3.org/2002/07/owl#ObjectProperty"))

f-4 (MAIN::triple (predicate "http://www.w3.org/2000/01/rdf-schema#range")
                  (subject "http://www.owlntologies.com/2010/Ontology1270064149.owl#CanCause")
                  (object "http://www.w3.org/2002/07/owl#IDATBOZB"))
```

Figure 5.13: Triple facts

EventExtraction.clp

The asserted facts to the working memory contain all the ontology annotation information, but we need only the information about events. Therefore, via EventExtraction file, we extract only facts that give information about event causes and consequences which have the "CanCause" predicate. For example the following fact should be extracted:

Without namespace declarations:

(triple (predicate "CanCause")
(subject " Vessel_rupture")
(object "leaking"))

With namespace declarations:

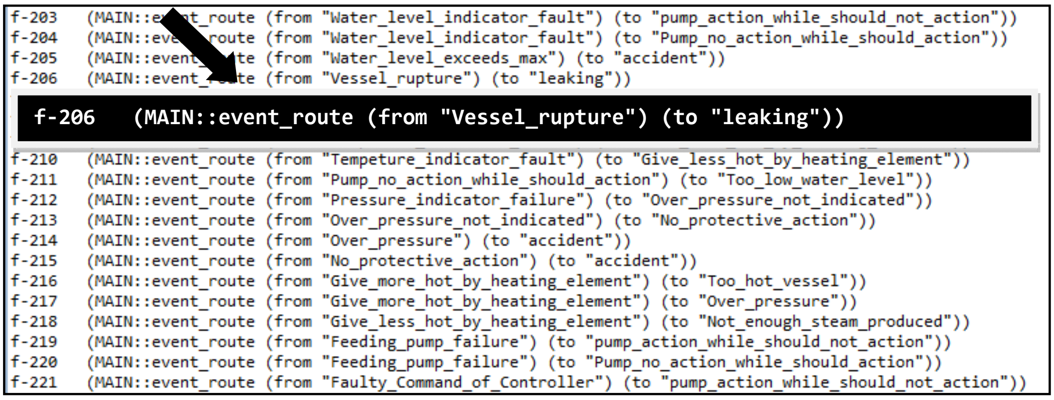
(triple (predicate "http://www.owl-ontologies.com/2010/Ontology1.owl#CanCause")
(subject "http://www.owl-ontologies.com/2010/Ontology1.owl# Vessel_rupture")
(object "http://www.owl-ontologies.com/2010/Ontology1.owl#leaking"))

This fact explains: Vessel_rupture → CanCause → leaking. When all facts with CanCause predicate are extracted, the extracted facts are converted to other facts and all the namespace declarations are removed. Forward chaining can be implemented easier over these facts than the triple PSO facts. For instance the above fact converts to the following fact and all the namespace declarations are removed:

(event_route (from " Vessel_rupture") (to "leaking"))

Figure 5.14 illustrates the result of this event facts extraction.

Ontologies and their elements are identified using Internationalized Resource Identifiers (IRIs) [20]. The ontology IRI identifies a particular version from an ontology series – the set of all the versions of a particular ontology identified using a common ontology IRI. The "IRI" of the related ontology should be entered in this file (in the specified location in the code). The ontology IRI should be inserted only the first time that we want to use a selected ontology for generating event chains. When we want to update the ontology and then the chains, it is not required to insert the ontology IRI again.



```
f-203    (MAIN::event_route (from "Water_level_indicator_fault") (to "pump_action_while_should_not_action"))
f-204    (MAIN::event_route (from "Water_level_indicator_fault") (to "Pump_no_action_while_should_action"))
f-205    (MAIN::event_route (from "Water_level_exceeds_max") (to "accident"))
f-206    (MAIN::event_route (from "Vessel_rupture") (to "leaking"))
f-206    (MAIN::event_route (from "Vessel_rupture") (to "leaking"))
f-210    (MAIN::event_route (from "Tempeture_indicator_fault") (to "Give_less_hot_by_heating_element"))
f-211    (MAIN::event_route (from "Pump_no_action_while_should_action") (to "Too_low_water_level"))
f-212    (MAIN::event_route (from "Pressure_indicator_failure") (to "Over_pressure_not_indicated"))
f-213    (MAIN::event_route (from "Over_pressure_not_indicated") (to "No_protective_action"))
f-214    (MAIN::event_route (from "Over_pressure") (to "accident"))
f-215    (MAIN::event_route (from "No_protective_action") (to "accident"))
f-216    (MAIN::event_route (from "Give_more_hot_by_heating_element") (to "Too_hot_vessel"))
f-217    (MAIN::event_route (from "Give_more_hot_by_heating_element") (to "Over_pressure"))
f-218    (MAIN::event_route (from "Give_less_hot_by_heating_element") (to "Not_enough_steam_produced"))
f-219    (MAIN::event_route (from "Feeding_pump_failure") (to "pump_action_while_should_not_action"))
f-220    (MAIN::event_route (from "Feeding_pump_failure") (to "Pump_no_action_while_should_action"))
f-221    (MAIN::event_route (from "Faulty_Command_of_Controller") (to "pump_action_while_should_not_action"))
```

Figure 5.14: Event facts extraction

ForwardChaining.clp

As described, forward chaining is one of the main methods of reasoning. For developing this file the forward chaining algorithm is used. In this file, previously created facts (extracted event cause-consequence facts from the transformed ontology) are used for reasoning. After running this file, the result would be a set of event chains which starts from an initial event (the initial event should be inserted by the user) to a final accident. Each chain may contain two or more events. Description for implementing the forward chaining algorithm is provided in Section 5.7.3.

5.7 A real world example

The developed system is tested with a real world example (Steam Boiler system); to see if the expected chains based on the inserted initial event can be generated with the prototype system. This section contains description of a simple steam boiler, testing the prototype system with the steam boiler example, forward chaining algorithm and the final result of the implemented test.

5.7.1 Steam boiler

A steam boiler is a device used to create steam by applying heat to water. A simple steam boiler contains components such as: vessel, heating element, controller, feeding pump, water level sensor and pressure sensor. The water level and steam pressure in the vessel should remain to specific values. The controller receives pressure readings from the sensors and adjusts the heater state. The pressure of the vessel should not exceed a predefined value. If steam pressure exceeds this predefined value, the controller should turn off the heater and the heater should be turned on if the steam pressure goes below it lowest value. Failure to pressure sensors can cause an explosion. Water level sensors control the water level in the vessel and sends signals to the controller. If water level exceeds the predefined value, the controller turns off the feeding pump. If water level goes below its lowest level, controller turns on the feeding pump. The steam boiler shown below is a simplified version of an industrial steam boiler. For having a simple system, important components such as the feeding tank and the blow down valve are left out [21]. Figure 5.15 is an illustration for a simple steam boiler system.

5.7.2 Testing the developed system with the steam boiler example

In order to examine the developed system and observe if it works as expected and the desired result can be produced, a test is performed. For implementing this test, we created an ontology with the specified format for the steam boiler system, in which an Event class is created, different event instances are added and then, information about direct connections between event cause to the consequence event is defined, an even instance to another event instance (Figure 5.16). Some of these direct connections between causes and consequences for the steam boiler system are presented here:

Vessel_rupture → leaking
leaking → accident
Pressure_indicator_failure → Over_pressure_not_indicated
Over_pressure_not_indicated → No_protective_action
No_protective_action → accident

These events are added to the ontology as event instances (individuals) of the "Event" class. Then, the related "CanCause" relationships are created between different event instances. For example:

Vessel_rupture CanCause leaking

Figure 5.17 illustrates a number of sequential events for the steam boiler system. Some of these sequential events cause an accident in the environment (1, 2, and 4). It is expected that the prototype system will be able to generate these chains after inserting the initial event; but the chains which do not terminate to an accident event should not be generated by the system (for example the third chain). After running the program, the user should insert the initial event. Then the system performs forward chaining over the Jess facts and the expected chains are created.

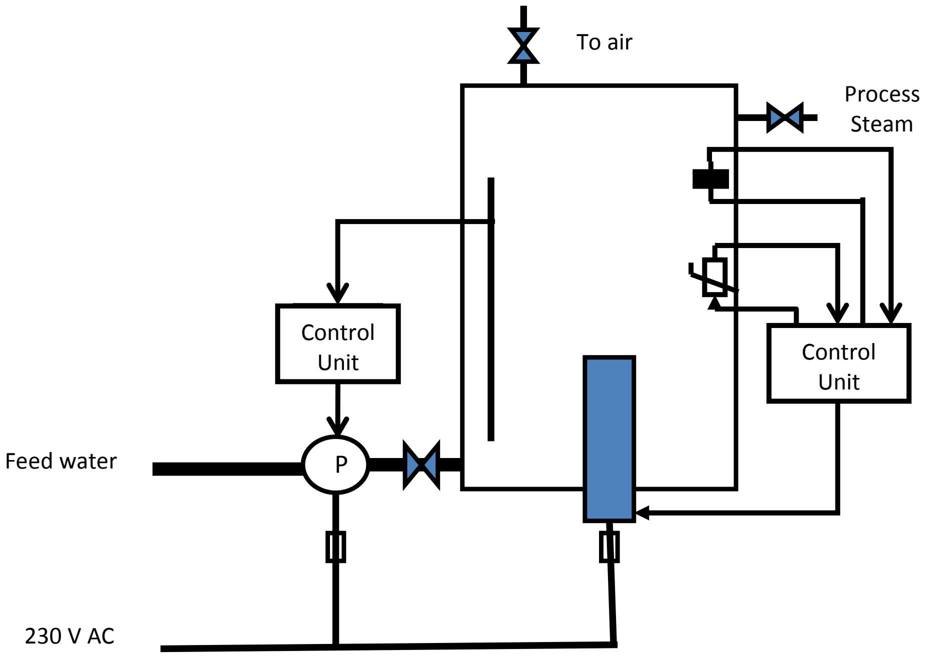


Figure 5.15: Illustration for a simple steam boiler system [21]

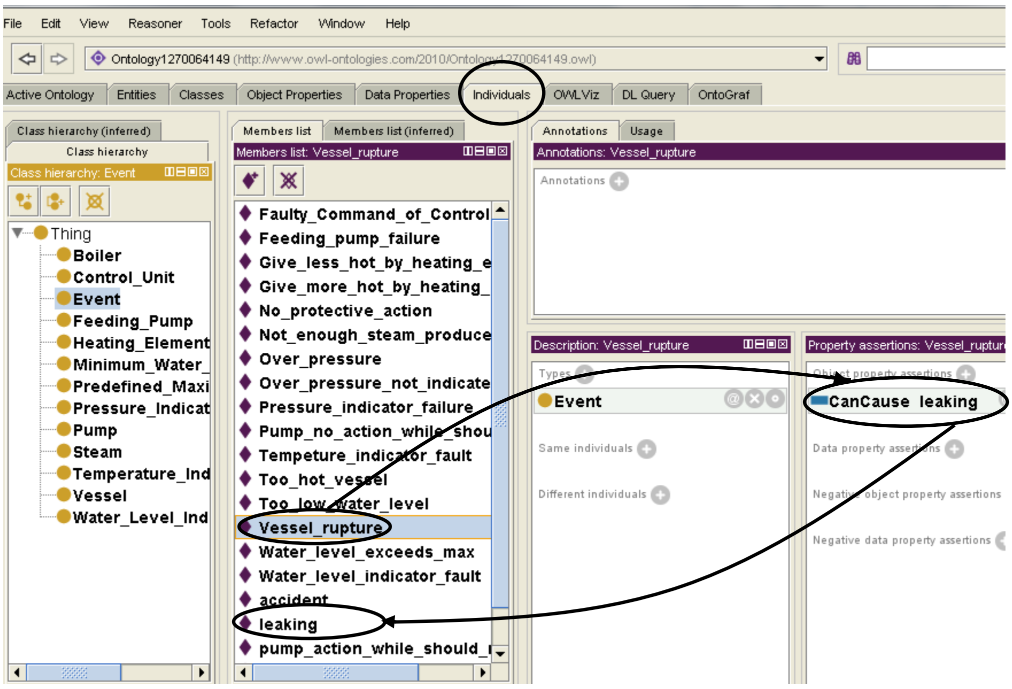

File Edit View Reasoner Tools Refactor Window Help
Ontology1270064149 (http://www.owl-ontologies.com/2010/Ontology1270064149.owl)
Active Ontology Entities Classes Object Properties Data Properties Individuals OWLViz DL Query OntoGraf
Class hierarchy (inferred)
Class hierarchy
Class hierarchy: Event
Thing
Boiler
Control_Unit
Event
Feeding_Pump
Heating_Element
Minimum_Water_
Predefined_Maxi
Pressure_Indicat
Pump
Steam
Temperature_Ind
Vessel
Water_Level_Ind
Members list
Members list (inferred)
Members list: Vessel_rupture
Faulty_Command_of_Control
Feeding_pump_failure
Give_less_hot_by_heating_e
Give_more_hot_by_heating_
No_protective_action
Not_enough_steam_produce
Over_pressure
Over_pressure_not_indicate
Pressure_indicator_failure
Pump_no_action_while_shou
Tempeture_indicator_fault
Too_hot_vessel
Too_low_water_level
Vessel_rupture
Water_level_exceeds_max
Water_level_indicator_fault
accident
leaking
pump_action_while_should_
Annotations
Usage
Annotations: Vessel_rupture
Annotations
Description: Vessel_rupture
Types
Event
Same individuals
Different individuals
Property assertions: Vessel_rupture
Object property assertions
canCause leaking
Data property assertions
Negative object property assertions
Negative data property assertions

Figure 5.16: Steam boiler ontology which contains cause-consequence concepts

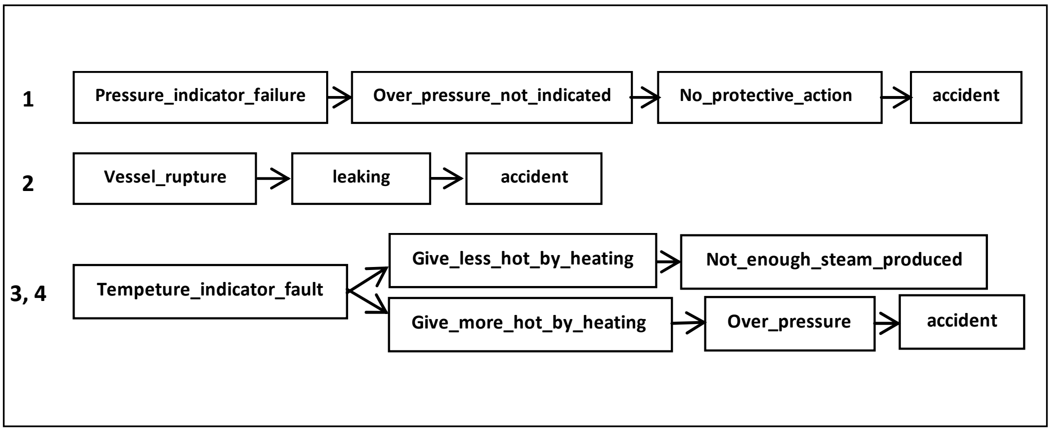


Figure 5.17: Some event chains for steam boiler system

5.7.3 Forward chaining algorithm [9]

Forward chaining is implemented by defining a set of rules and applying these rules to the Jess facts. These rules are in the following format:
If LHS **Then** RHS

For example:
(defrule R1   (A) => (assert (B)))
R1: If A is in working memory then assert B to working memory

Rules fire when their LHS are satisfied or matched by facts in the working memory. When rules fire, additional knowledge is gained. The consequent of the rules may be an assertion of new facts to the working memory. When rules fire, they may create a situation where other rules can fire as well. This form of inference is called forward chaining [9]. In this project, we need to perform forward chaining over cause–consequence facts. These facts give Information about direct connections between events:

Cause event → Consequence event
event A → Event B

The facts are asserted to the working memory in the following form:

event_route (from "**event A**")(to "**event B**")

For example:
event_route (from "**Tempeture_indicator_fault**") (to "**Give_more_hot_by_heating_element**")
event_route (from "**Give_more_hot_by_heating_element**") (to "**Over_pressure**")
event_route (from "**Over_pressure**") (to "**accident**")

A set of rules is then defined to generate chains from initial event to accident. A fact in which the initial event (inserted by the user) and the final event(accident) are stored, should be created. The following template is used for this purpose:

(deftemplate **my_event_chain** (slot **InitialEvent**)(slot **FinalEvent**))

The "**FinalEvent**" slot is set to "accident", because we want to generate chains with accident as the final event. When the user inserts the initial event, this event is stored as the "**InitialEvent**" slot in the "**my_event_chain**" template and a new fact asserts to the working memory. For example:

my_event_chain (InitialEvent "Feeding_pump_failure") (FinalEvent "accident")

After inserting the initial event, the following rules are executed over the "event_route" facts to test whether a chain can be created from the initial event to accident. These rules are taken from the "Flight Advisor" example in [9], but some modifications are made in the rules for using them in the event chains generation system.

Rule 1:

(defrule **from-start**
        (**my_event_chain** (**InitialEvent** ?start))
        (**event_route** (**from** ?start)(**to** ?intermediate-location))
=>
(assert (**reachable** (**from** ?start)(**to** ?intermediate-location))))

Based on this rule if any "event_route" facts in the working memory exist which have the "from" slot as the inserted initial event, then new facts will be asserted to the working memory. These new asserted facts correspond to the following template:

(deftemplate **reachable**(slot **from**)(slot **to**))

and gives information about all consequence events that occurred because our initial event is occurred. For example Figure 5.18 presents a set of event route facts. It can be seen that applying rule 1 to these facts causes two activations, B and C.

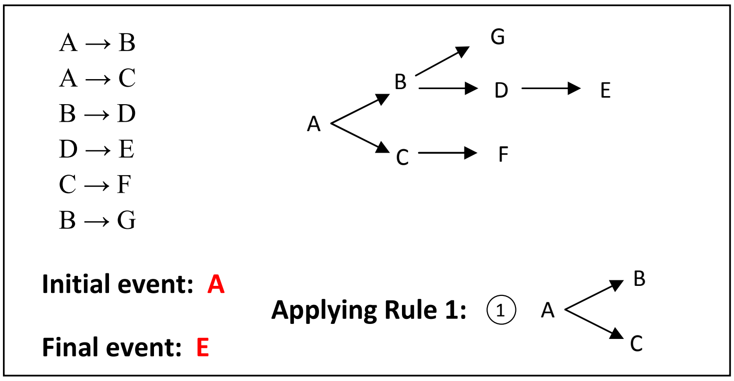


Figure 5.18: Applying Rule 1 to the facts

Rule 2:

(defrule **to-intermediate**

  (**reachable** (**from** ?start)(**to** ?intermediate-location1))

  (**event_route** (**from** ?intermediate-location1)(**to** ?intermediate-location2))

=>

(assert (**reachable** (**from** ?intermediate-location1)(**to** ?intermediate-location2))))

This rule uses our new facts created in the rule 1 and our event_route information for creating all the outgoing events from any of these found events. Applying rule 2 to the previous example (Figure 5.18) would result in two phase of activations. It can be seen in Figure 5.19 that after applying rule 2, all the events that had the root B or C have been detected

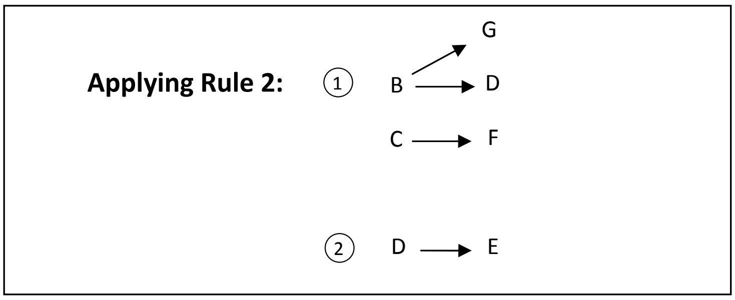


Figure 5.19: Applying Rule 2 to the facts

Rule 3:

(defrule **to-destination**

  (**my_event_chain** (**InitialEvent** ?start)(**FinalEvent** ?destination))

  (**reachable** (**from** ?start)(**to** ?intermediate-location1))

  (**event_route** (**from** ?intermediate-location1)(**to** ?destination))

=>

(printout t " (" ?start ") → (" ?intermediate-location1 ") → (" ?destination") " crlf))

This rule tests whether any of new asserted facts ("reachable" facts) has the "from" slot as the inserted initial event and if there is any "event_route" facts that has the "to" slot as accident. If these two facts exist, then a chain can be generated and printed. The rule 3 prints a chain which contains three events. Similar to this rule, other rules are defined in the main program for printing chains with more events. Currently the system can print event chains with limited number of events in a chain. The rules for creating all connections in a chain are already defined; therefore we only require defining a rule which enables printing unlimited number of events in a chain. Figure 5.20 presents after applying rule 3, the desired event routes from the initial event to the final event are identified.

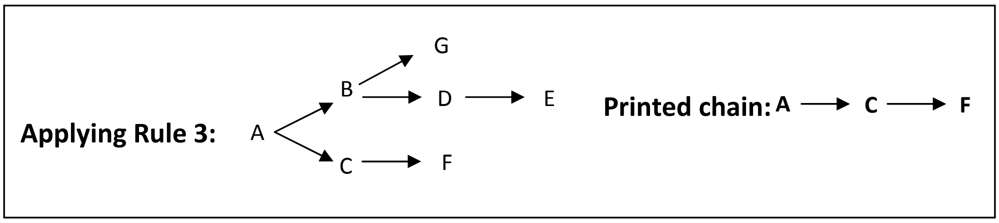


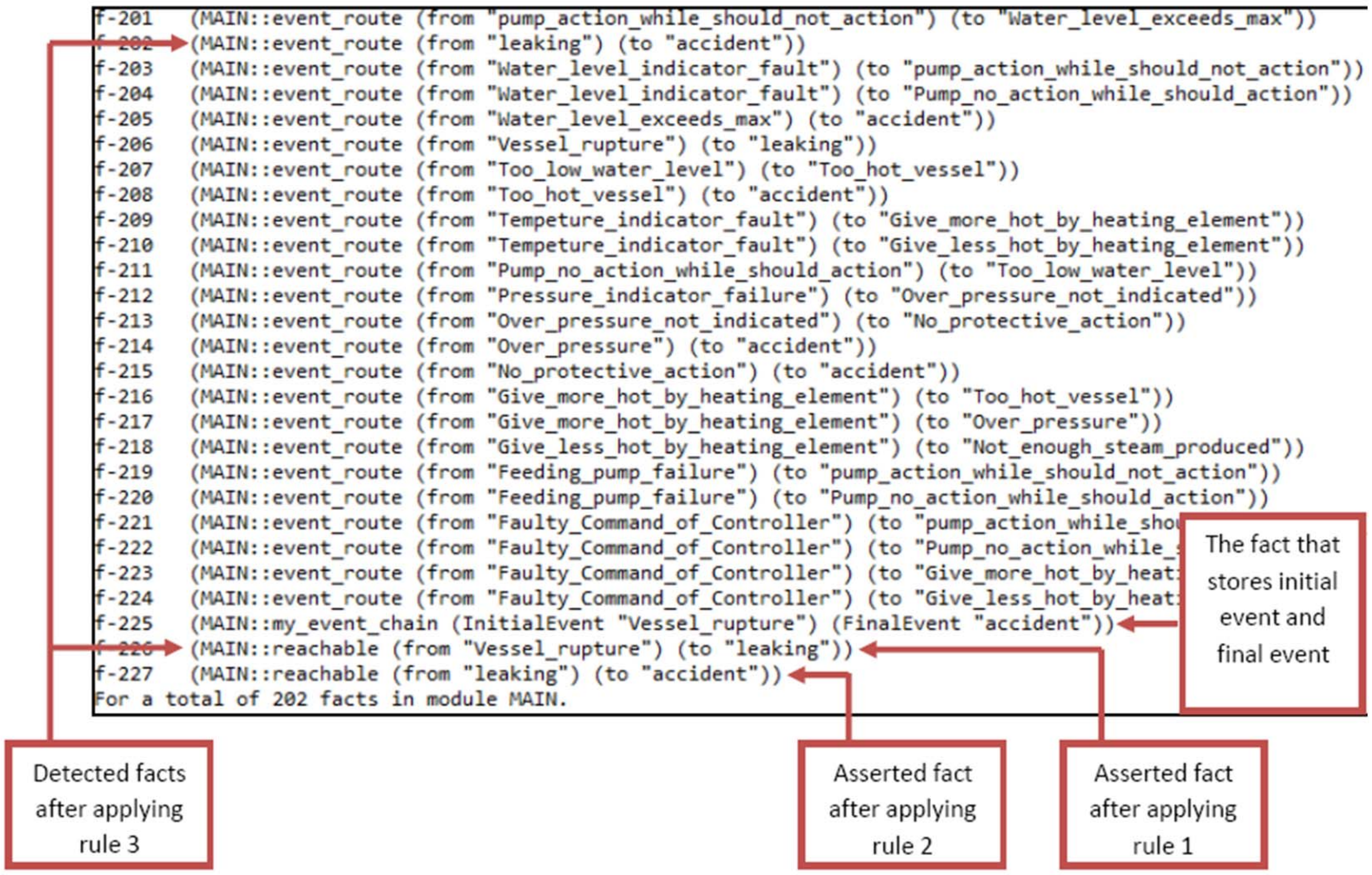
Figure 5.20: Applying Rule 3 to the facts

5.7.4 Final result

The result of the test was generation of a set of event chains. As expected forward chaining performed over Jess facts correctly and the rules are activated and fired truly. Figure 5.21 shows an example of rule activations for steam boiler example, after inserting the initial event "Vessel_rupture". Finally the following chain will be generated:

Vessel_rupture→ leaking → accident

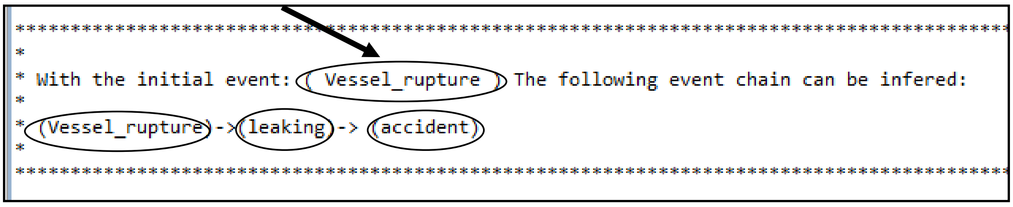
Figure 5.22 presents this event chain that is printed to the Eclipse console view after inserting the initial event "Vessel_rupture". More tests performed to the system. The results of these tests with some snapshots are provided in Appendix C.



```
f-201   (MAIN::event_route (from "pump_action_while_should_not_action") (to "Water_level_exceeds_max"))
f-202   (MAIN::event_route (from "leaking") (to "accident"))
f-203   (MAIN::event_route (from "Water_level_indicator_fault") (to "pump_action_while_should_not_action"))
f-204   (MAIN::event_route (from "Water_level_indicator_fault") (to "Pump_no_action_while_should_action"))
f-205   (MAIN::event_route (from "Water_level_exceeds_max") (to "accident"))
f-206   (MAIN::event_route (from "Vessel_rupture") (to "leaking"))
f-207   (MAIN::event_route (from "Too_low_water_level") (to "Too_hot_vessel"))
f-208   (MAIN::event_route (from "Too_hot_vessel") (to "accident"))
f-209   (MAIN::event_route (from "Tempeture_indicator_fault") (to "Give_more_hot_by_heating_element"))
f-210   (MAIN::event_route (from "Tempeture_indicator_fault") (to "Give_less_hot_by_heating_element"))
f-211   (MAIN::event_route (from "Pump_no_action_while_should_action") (to "Too_low_water_level"))
f-212   (MAIN::event_route (from "Pressure_indicator_failure") (to "Over_pressure_not_indicated"))
f-213   (MAIN::event_route (from "Over_pressure_not_indicated") (to "No_protective_action"))
f-214   (MAIN::event_route (from "Over_pressure") (to "accident"))
f-215   (MAIN::event_route (from "No_protective_action") (to "accident"))
f-216   (MAIN::event_route (from "Give_more_hot_by_heating_element") (to "Too_hot_vessel"))
f-217   (MAIN::event_route (from "Give_more_hot_by_heating_element") (to "Over_pressure"))
f-218   (MAIN::event_route (from "Give_less_hot_by_heating_element") (to "Not_enough_steam_produced"))
f-219   (MAIN::event_route (from "Feeding_pump_failure") (to "pump_action_while_should_not_action"))
f-220   (MAIN::event_route (from "Feeding_pump_failure") (to "Pump_no_action_while_should_action"))
f-221   (MAIN::event_route (from "Faulty_Command_of_Controller") (to "pump_action_while_shou
f-222   (MAIN::event_route (from "Faulty_Command_of_Controller") (to "Pump_no_action_while_s
f-223   (MAIN::event_route (from "Faulty_Command_of_Controller") (to "Give_more_hot_by_heat
f-224   (MAIN::event_route (from "Faulty_Command_of_Controller") (to "Give_less_hot_by_heat
f-225   (MAIN::my_event_chain (InitialEvent "Vessel_rupture") (FinalEvent "accident"))
f-226   (MAIN::reachable (from "Vessel_rupture") (to "leaking"))
f-227   (MAIN::reachable (from "leaking") (to "accident"))
For a total of 202 facts in module MAIN.
```

The fact that stores initial event and final event
Detected facts after applying rule 3
Asserted fact after applying rule 2
Asserted fact after applying rule 1

Figure 5.21: Rule activations for the steam boiler example



```
*******************************************************************************************
*
* With the initial event: ( Vessel_rupture ) The following event chain can be infered:
*
* (Vessel_rupture)->(leaking)-> (accident)
*
*******************************************************************************************
```

Figure 5.22: Generated evens chain after inserting the initial event

6 Conclusions and further works

In this chapter we first present summary and some concluding comments of this project. After this, we discuss some further work.

6.1 Conclusions

The goal of this project was to facilitate safety analysis by developing a system model which enables semi-automatically generation of event chains for a selected domain using an ontology of the domain which contains safety expert knowledge (cause-consequence concepts). The following points summarize the outcomes of this project.

To achieve the goal, related information and methods were collected and studied. During this information gathering, we learned that a semantic reasoner should be used for developing a system which can reason about cause-consequence concepts of a domain and generates related event chains. We then performed a survey of available methods for reasoning. The survey identified several methods of reasoning which could be used in this project. After reviewing these methods, the forward chaining method was selected for reasoning due to its particular features - which was described in Chapter 3 - such as its implementation simplicity.

Safety expert knowledge containing information on previous accident experiences and the cause-consequence concepts in a specific domain stored as an ontology. The semantic reasoner (Jess) used this ontology for reason about. For implementing the forward chaining algorithm for reasoning over the ontology, a set of rules are defined in the Eclipse platform. The result of applying these rules over the ontology was the generation of sequential events (event chains) which end with an accident event. Automatically identifying event chains can facilitate safety analysis for a complex system, reduce the amount of faults that results from manually identifying sequential events, increase the accuracy of safety analysis and enables (re)use of safety expert knowledge. Thus, when event chains that end with an accident event are identified, we can change the system so that we can prevent them from occurring.

This system model implemented the necessary ontology for a real world example (steam boiler).The causes and consequences concepts of the steam boiler system were added to its pre developed ontology. This ontology file was then transformed for using in a Rule Engine (Jess). Rules for implementing the forward chaining algorithm were defined and applied for reasoning about causes and consequences concepts in the transformed ontology. The result of this implementation was acceptable and the expected event chains were generated automatically. Snapshots for this implementation can be found in the Appendix C.

Other benefits of using this system are:

1) Updating event chains after ontology modifications is easy. This updating may result in the generation of new chains or modifications to the previously generated chains,

while manually tracking event sequences after a small modification to the ontology for a large system is difficult. For example, if the event cause-consequence connections modified in the ontology: (A CanCause B) is modified to (A CanCause C), the related event chains will be updated automatically after transforming the modified ontology to the Jess usable format.

2) Multiple ontologies can be used simultaneously for reasoning. Related chains will be generated based on the inserted ontologies. This feature would be useful when we need to use more than one ontology simultaneously for safety analysis, for example a domain ontology and an environment ontology

The developed system will facilitate safety analysis by effective (re)use of safety expert knowledge in the form of ontologies and improving safety knowledge of the selected domain.

6.2 Further works

Enhancing the event chains generation system with some features will increase its benefits for safety analysis. Some of these features are explained in this section and considered as further work of this project.

1) The current event chains generation system is capable of generating chains which start from an initial event and end with an accident event.

   Initial event → ……... → accident

   But in the real world two independent events may cause a common accident (A or B) or in another situation, two events occurring at the same time cause an accident (A and B). The developed chains for these cases would be like following:

   [(event A) **and (**event B)] → ……... → accident
   [(event A) **or (**event B)] → ……... → accident

   Implementing these features requires adding logical rules for reasoning to the current system and defining these rules is not complex.

2) In a real world situation, the (A CanCause B) relation can be conditional on a third event. This means that the event A can cause the event B if the event C already has taken place. The system should be able to generate the relevant chains based on existed conditions. For example the following connection will be part of an event chain if its condition ("C") is satisfied: ((A CanCause B) if C). For this purpose, our idea is to add this conditional information - for example information about dependency of the (A CanCause B) relation to the event "c" - to the ontology which all the cause-consequence concepts are stored. The reasoning must then be performed based on these conditions. If the condition for a CanCause connection satisfied, this connection will be used for forward chaining.

3) The enhanced system can import ontologies which can be used for reasoning via its GUI. By enhancing the system with this feature, the OWL transformation to Jess assertions will be performed automatically. Currently a software tool is developed which enables automatically transforming OWL ontologies to COOL language of CLIPS [22]. This software performs the transformation task very well, but the transformed file should be modified in order to be usable with the Jess inference system.

4) The current system can print event chains with a limited number of events in a chain. The rules for creating all connections in a chain are already defined; therefore we only need to define a rule which enables printing an unlimited number of events in a chain (Figure 6.1). This rule can use the event connections created from other defined rules, for printing.

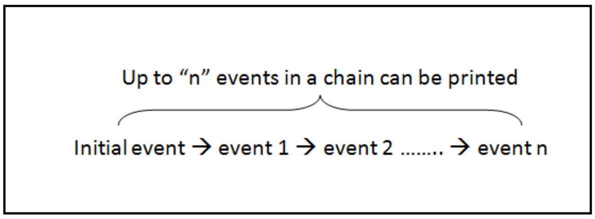

Up to "n" events in a chain can be printed
Initial event → event 1 → event 2 .... → event n

Figure 6.1: A chain with "n" events

5) The GNLQ tool is a knowledge-based, guided requirements elicitation environment. Since the GNLQ tool currently has some capabilities for safety analysis, we should see if it is possible to upgrade this tool with the chaining algorithms that are used for developing the event chains generation system. If this upgrading is possible, the GNLQ tool is not only a knowledge-based guided requirements elicitation environment, it will become also a knowledge based safety analysis environment.

Bibliography

[1] Silvianita, Mohd. Faris Khamidi, Kurian V. John, “Critical Review of a Risk Assessment Method and its Applications”, International Conference on Financial Management and Economics, Singapore, 2011.

[2] Tor Stålhane, Stefan Farfeleder, Olawande Daramola, "Safety analysis based on requirements", Norwegian University of Science and Technology and Vienna University of Technology, 2011.

[3] Inah Omoronyia and Tor Stålhane, “Guided Natural Language and Requirement Boilerplates”, TDT4242 Requirements and testing, Trondheim, Norway, 2012.

[4] Tor Stålhane, “Safety Analysis and Boilerplates”, Presented in Technical University of Wien and IFE - Halden, Wien, Austra, 27 November 2012.

[5] National Institute of General Medical Sciences, (2013) “protégé website”, [Online] Available from: http://protege.stanford.edu/ [Accessed 25.04.13].

[6] Natalya F. Noy and Deborah L. Mc Guinness, “Ontology Development 101: A Guide to Creating Your First Ontology”, Stanford University, [Online] Available from: http://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html [Accessed 20.03.13].

[7] The JBoss Drools team, “Drools Expert User Guide”, [Online] Available from: http://docs.jboss.org/drools/release/5.2.0.Final/drools-expert-docs/html/index.html [Accessed 25.04.13].

[8] Robinson Selvamony, “Introduction to the Rete Algorithm”, SAP Labs, India, 17 December 2010.

[9] Martin J. Kollingbaum, “Revision of Jess”, Programming Expert Systems with Jess CS3019, Knowledge-Based Systems, Lecture 21, University of Aberdeen, United Kingdom.

[10] Charles L. Forgy, “Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*”, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, U.S.A, April 1981.

[11] Yong-Hua Song, Allan Johns and Raj Aggarwal, Expert systems: An introduction. In “Computational Intelligence Applications to Power Systems”, Beijing, Science Press and Kluwer Academic, pp. 8-23.

[12] Micheal Hewett, (June 06, 2005) "Algernon - Rule-Based Programming", [Online] Available from: http://algernon-j.sourceforge.net/ [Accessed 10.05.13].

[13] Ernest J. Friedman-Hill, Sandia National Laboratories, (5 November 2008) "Jess® the Rule Engine for the Java™ Platform", Version 7.1p2, [Online] Available from: http://www.jessrules.com/jess/docs/71/index.html [Accessed 25.04.13].

[14] Henrik Eriksson, "JessTab Tutorial", Presented in the 11th International Protégé Conference, Amsterdam, Netherland, 23 June 2009.

[15] Fabien L. Gandon and Norman M. Sadeh, (July 16, 2003) "OWL inference engine using XSLT and JESS", [Online] Available from: http://mcom.cs.cmu.edu/OWL/OWLEngine.html [Accessed 01.04.13].

[16] Leonid Kof, "Using Application Domain Ontology to Construct an Initial System Model", Faculty of computer science, Technical University of Munich, Boltzmannstr, Munich, Germany, 2004.

[17] L. Kof, "An Application of Natural Language Processing to Requirements Engineering — A Steam Boiler Case Study", Contribution to ICSE, 2004.

[18] Safoura Shamsolketabi and Majid Navaii, "An overview on Protégé manual", Department of Computer and Information Science, NTNU, Trondheim, Norway, August 2012.

[19] Ernest J. Friedman-Hill, Sandia National Laboratories, (Last modified: 12 Oct 2012) "Jess® the Rule Engine for the Java™ Platform", [Online] Available from: http://www.jessrules.com/ [Accessed 10.05.13].

[20] The World Wide Web Consortium (W3C), (11 December 2012) "OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)", [Online] Available from: http://www.w3.org/TR/owl2-syntax/ [Accessed 01.04.13].

[21] Tor Stålhane[1], Stefan Farfeleder[2], Olawande Daramola[1], "Safety analysis based on requirements", Norwegian University of Science and Technology[1] and Vienna University of Technology[2], 2010.

[22] Georgios Meditskos, Nick Bassiliades, "Towards an Object-Oriented Reasoning System for OWL", Department of Informatics, Aristotle University of Thessaloniki, Greece, 2005.

Appendix A – Jess Rule Engine requirements [19]

This appendix contains an overview of requirements that must be available for using the Jess. Jess is a programmer's library written in Java. Therefore, to use Jess, you'll need a Java Virtual Machine (JVM). Be sure your JVM is installed and working correctly before trying to use Jess. To use the Jess DE integrated development environment, you'll need version 3.1 or later of the Eclipse SDK from http://www.eclipse.org. Be sure that Eclipse is installed and working properly before installing the Jess DE. The Jess language is a highly specialized form of Lisp. The user must have a Java runtime system and know how to use it to:

- run a Java application
- deal with configuration issues like the CLASSPATH variable
- (optional) compile a collection of Java source files

Also the user must have general familiarity with the principles of programming.

Appendix B – Developed codes

This appendix contains all developed codes in this project. The codes are developed in three files:

1) OntologyTransformation.clp
2) EventExtraction.clp
3) ForwardChaining.clp

The first part of the code in "OntologyTransformation.clp" is the "OWL Meta model" which is taken from [15]. This OWL meta model defines templates that exist in an OWL ontology.

OntologyTransformation.clp

```
;;; OWL Meta model ---------------------------------------------------------------

;;; Declaring the triple template --------------------------------
(deftemplate triple "Template representing a triple"
 (slot predicate (default ""))
 (slot subject   (default ""))
 (slot object    (default ""))
)

;;; Declaring facts of the meta-model of OWL ---------------------
(deffacts OWLModel "Facts of the meta-model of RDFS and OWL"

  ;;; Resource is a Class
  (triple
   (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
   (subject   "http://www.w3.org/2000/01/rdf-schema#Resource")
   (object    "http://www.w3.org/2002/07/owl#Class")
  )

  ;;; Class is a subtype of Resource
  (triple
   (predicate "http://www.w3.org/2000/01/rdf-schema#subClassOf")
   (subject   "http://www.w3.org/2002/07/owl#Class")
   (object    "http://www.w3.org/2000/01/rdf-schema#Resource")
  )
   ;;; Class is a Class
```

```
(triple
 (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
 (subject   "http://www.w3.org/2002/07/owl#Class")
 (object    "http://www.w3.org/2002/07/owl#Class")
)

;;; Thing is a Class
(triple
 (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
 (subject   "http://www.w3.org/2002/07/owl#Thing")
 (object    "http://www.w3.org/2002/07/owl#Class")
)

;;; Nothing is a Class
(triple
 (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
 (subject   "http://www.w3.org/2002/07/owl#Nothing")
 (object    "http://www.w3.org/2002/07/owl#Class")
)

;;; Property is a Class
(triple
 (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
 (subject   "http://www.w3.org/1999/02/22-rdf-syntax-ns#Property")
 (object    "http://www.w3.org/2002/07/owl#Class")
)

;;; Domain is a Property
(triple
 (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
 (subject   "http://www.w3.org/2000/01/rdf-schema#domain")
 (object    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Property")
)

;;; Range is a Property
(triple
 (predicate "http://www.w3.org/2000/01/rdf-schema#range")
 (subject   "http://www.w3.org/2000/01/rdf-schema#domain")
 (object    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Property")
)

;;; 'Data Type Property' is a Class
(triple
 (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
 (subject   "http://www.w3.org/2002/07/owl#DatatypeProperty")
 (object    "http://www.w3.org/2002/07/owl#Class")
)
```

```
 ;;; 'Object Property' is a Class
 (triple
  (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
  (subject   "http://www.w3.org/2002/07/owl#ObjectProperty")
  (object    "http://www.w3.org/2002/07/owl#Class")
 )

 ;;; 'Object Property' is a subtype of Property
 (triple
  (predicate "http://www.w3.org/2000/01/rdf-schema#subClassOf")
  (subject   "http://www.w3.org/2002/07/owl#ObjectProperty")
  (object    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Property")
 )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

;;; The transformed ontology annotations should be entered here:

```
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2010/Ontology1270064149.owl")
                (object "http://www.w3.org/2002/07/owl#Ontology") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#delivers")
                (object "http://www.w3.org/2002/07/owl#ObjectProperty") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#has_a")
                (object "http://www.w3.org/2002/07/owl#ObjectProperty") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#CanCause")
                (object "http://www.w3.org/2002/07/owl#ObjectProperty") ) )
 (assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                 (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#Feeding_Pump")
                 (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#Heating_Element")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com.owl#Minimum_Water_Level")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/.owl#Predefined_Maximum_Water_Level")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#Pressure_Indicator")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#Pump")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
```

```
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.SteamBoiler.owl#Temperature_Indicator")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2009/SteamBoiler.owl#Vessel")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.w3.org/2002/07/owl#IDAFFIIB")
                (object "http://www.w3.org/2002/07/owl#Restriction") ) )
(assert (triple (predicate "http://www.w3.org/2002/07/owl#onProperty")
                (subject "http://www.w3.org/2002/07/owl#IDAFFIIB")
                (object "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#CanCause")
) ) (assert (triple (predicate "http://www.w3.org/2002/07/owl#allValuesFrom")
                (subject "http://www.w3.org/2002/07/owl#IDAFFIIB")
                (object "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#Event") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#Steam")
                (object "http://www.w3.org/2002/07/owl#Class") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                 (subject "http://www.owl-
ontologies.com/2010/.owl#Faulty_Command_of_Controller")
                (object "http://www.w3.org/2002/07/owl#NamedIndividual") ) )
 (assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                 (subject "http://www.owl-
ontologies.com.owl#pump_action_while_should_not_action")
                 (object "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#Event") ) )
(assert (triple (predicate "http://www.owl-
ontologies.com/2010/Ontology1270064149.owl#CanCause")
                (subject "http://www.ontologies.com/2010 #pump_action_while_should_not_action")
                (object "http://www.owl-ontologies.com/.owl#Water_level_exceeds_max") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ntologies.com/2010/Ontology12.owl#Vessel_rupture")
                (object "http://www.w3.org/2002/07/owl#NamedIndividual") ) )
(assert (triple (predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type")
                (subject "http://www.owl-ontologies.com/2010/Ontology19.owl#Vessel_rupture")
                (object "http://www.owl-ontologies.com/2010/Ontology1270064149.owl#Event") ) )
;;; EOF
```

EventExtraction.clp

```
(require* OntologyTransformation)

;;; Search for predicates that contain "CanCause".
(defquery search-by-predicate
  "Finds triples with hasbase predicate"
  (declare (variables ?predicate))
  (triple (predicate ?predicate) (subject ?subject) (object ?object)))
```

;;; Each ontology document can be accessed via an IRI

```
;;;The "IRI" part of the related ontology should be entered here:
(bind ?result (run-query* search-by-predicate (str-cat "http://www.owl-
ontologies.com/2010/Ontology1270064149.owl" "#CanCause" )))

(deftemplate event_route1 (slot from1) (slot to1))
(while (?result next)
        (assert (event_route1 (from1 (?result getString subject)) (to1 (?result getString object)))))
```

;;; Asserted facts from the ontology are in the IRI format, therefore the "Event" keywords should be extracted from it.

```
(deftemplate event_route (slot from)(slot to))
(defrule upgrade_routes
 ?R <- (event_route1 (from1 ?a)(to1 ?b))
 =>
  (retract ?R)       ;we remove "event_route1" facts and replace "event_route" facts with required
keywords.
  (assert (event_route
              (from (sub-string (+ (str-index "#" ?a) 1) (str-length ?a) ?a))
              (to (sub-string (+ (str-index "#" ?b) 1) (str-length ?b) ?b)))
         )
   )
 (run)
;;; EOF
```

ForwardChaining.clp

```
(deftemplate event_route (slot from)(slot to))
(deftemplate my_event_chain (slot InitialEvent)(slot FinalEvent))
(deftemplate reachable(slot from)(slot to))
(deftemplate detected_chain (slot InitialEvent)(slot intermediate)(slot FinalEvent))
(watch all)
```

;;; Rule 1: create all chain legs for events reachable from our initial event.

```
(defrule from-start
  (my_event_chain (InitialEvent ?start))
  (event_route (from ?start)(to ?intermediate-location))
 =>
   (assert (reachable (from ?start)(to ?intermediate-location )))
   )

(require* EventExtraction)
;;;Get input function (Gets the initial event from the input)
(deffunction get-input()
            “Get user input from console.”
            (bind ?s (read))
            (return ?s))
            (printout t "Please insert the initial event : " crlf)
            (assert (my_event_chain (InitialEvent (get-input))(FinalEvent "accident"))
  )

;;; All cause-consequence links are created here!
;;; Rule 2: creates all chain legs for new events reachable from currently reachable
events
(defrule to-intermediate1
   (reachable (from ?start)(to ?intermediate-location1))
   (event_route (from ?intermediate-location1)(to ?intermediate-location2))
=>
            (assert (reachable (from ?intermediate-location1)(to ?intermediate-location2))))
(run)
;;; Rule 3, this rule finds event chains with only two event nodes from "Initial Event" to
"accident".
(defrule to-destination1
   (my_event_chain (InitialEvent ?start)(FinalEvent ?destination))
            (reachable (from ?start)(to ?destination))
=>
     (printout t "                                   " crlf)
     (printout t *******************************************************" crlf)
     (printout t "*                                                                                  " crlf)
     (printout t "* With the initial event: ( " ?start" ) The following event chain can be infered:
" crlf)
     (printout t "*                                                                                 " crlf)
     (printout t "* (" ?start   ")->("   ?destination")          " crlf)
     (printout t "*                                                                                 " crlf)
     (printout t "*******************************************************" crlf)
     (printout t "                                                                                " crlf)
  )
(run)

;;; Rule 4, this rule finds event chains with three event nodes.
(defrule to-destination2
   (my_event_chain (InitialEvent ?start)(FinalEvent ?destination))
            (reachable (from ?start)(to ?intermediate-location1))
            (event_route (from ?intermediate-location1)(to ?destination))
=>
     (printout t "                                   " crlf)
     (printout t "*******************************************************" crlf)
     (printout t "*                                                                                  " crlf)
     (printout t "* With the initial event: ( " ?start" ) The following event chain can be infered:
" crlf)
     (printout t "*                                                                                 " crlf)
```

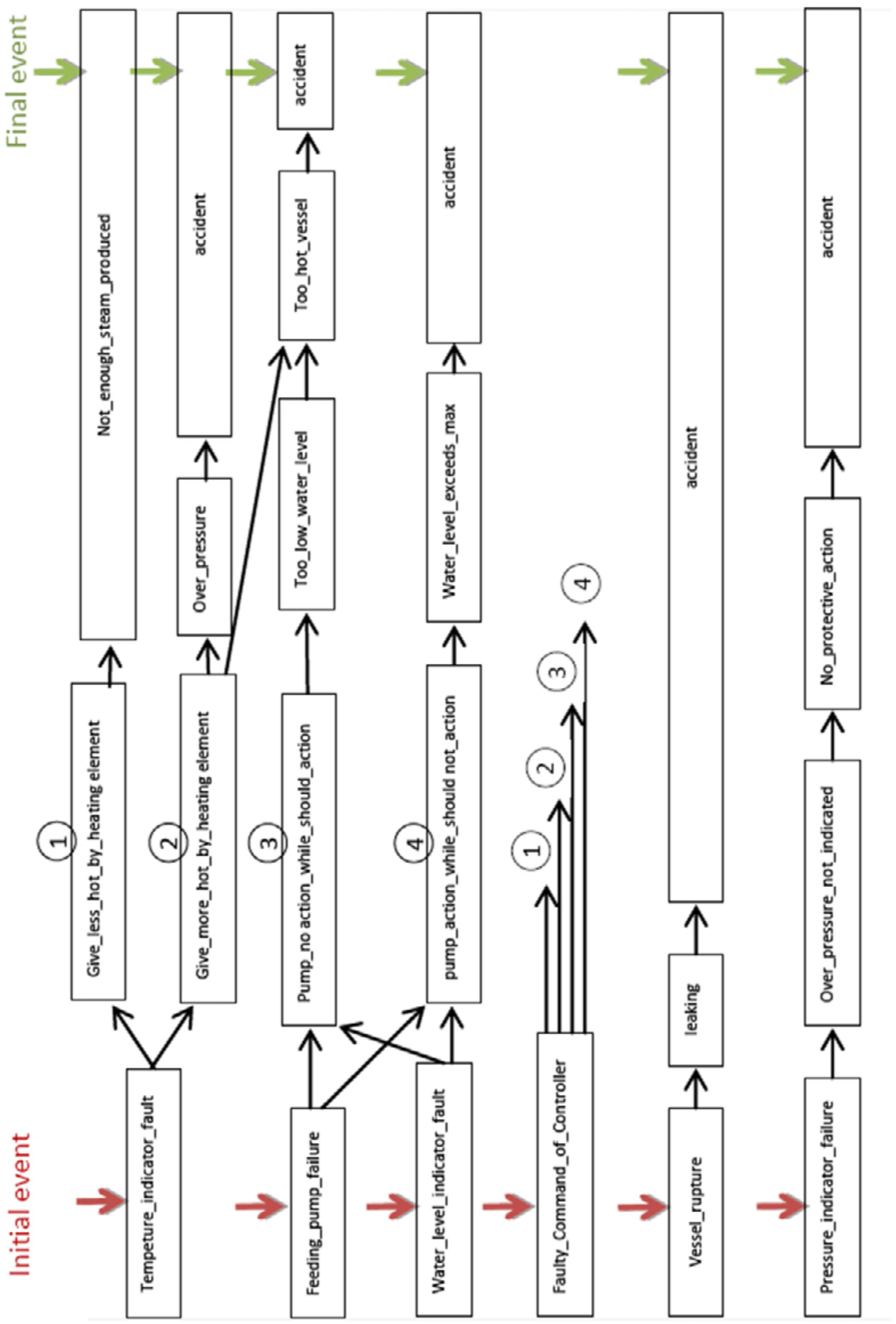
```
    (printout t "* (" ?start   ")->("   ?intermediate-location1   ")-> ("   ?destination")       " crlf)
    (printout t "*                                                              " crlf)
    (printout t "******************************************************" crlf)
    (printout t "                                                                " crlf)
  )

(run)
```

;;; Rule 5 , this rule finds event chains with four event nodes.

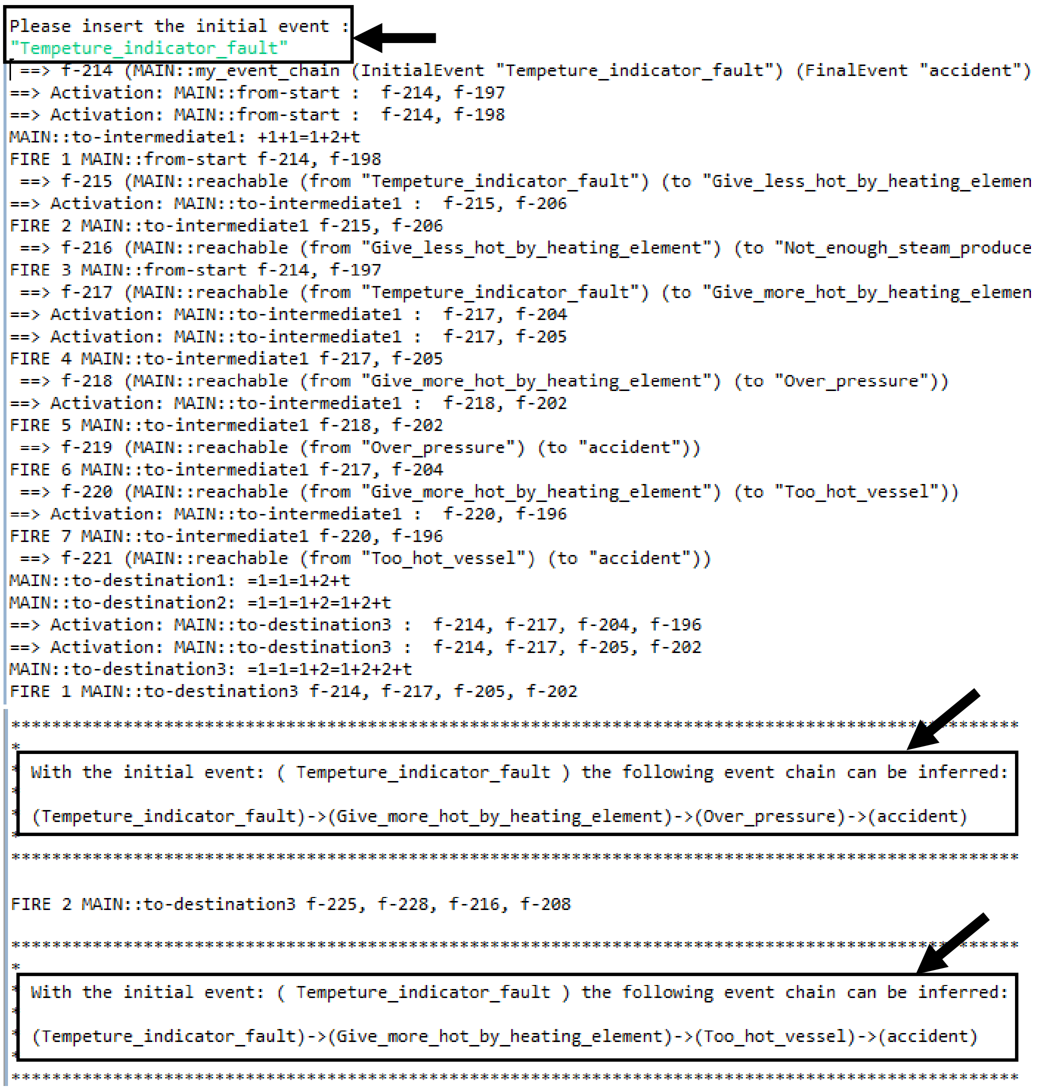
```
(defrule to-destination3
   (my_event_chain (InitialEvent ?start)(FinalEvent ?destination))
          (reachable (from ?start)(to ?intermediate-location1))
          (event_route (from ?intermediate-location1)(to ?intermediate-location2))
   (event_route (from ?intermediate-location2)(to ?destination))
=>
    (printout t "                              " crlf)
    (printout t "******************************************************" crlf)
    (printout t "*                                                              " crlf)
    (printout t "* With the initial event: ( " ?start" ) The following event chain can be infered:
" crlf)
    (printout t "*                                                              " crlf)
    (printout t "* (" ?start   ")->("   ?intermediate-location1   ")->("   ?intermediate-location2   ")->("
?destination")       " crlf)
    (printout t "*                                                              " crlf)
    (printout t "******************************************************" crlf)
    (printout t "                                                                " crlf)
  )

 (facts)
(run)
;;; EOF
```

Appendix C – Test results

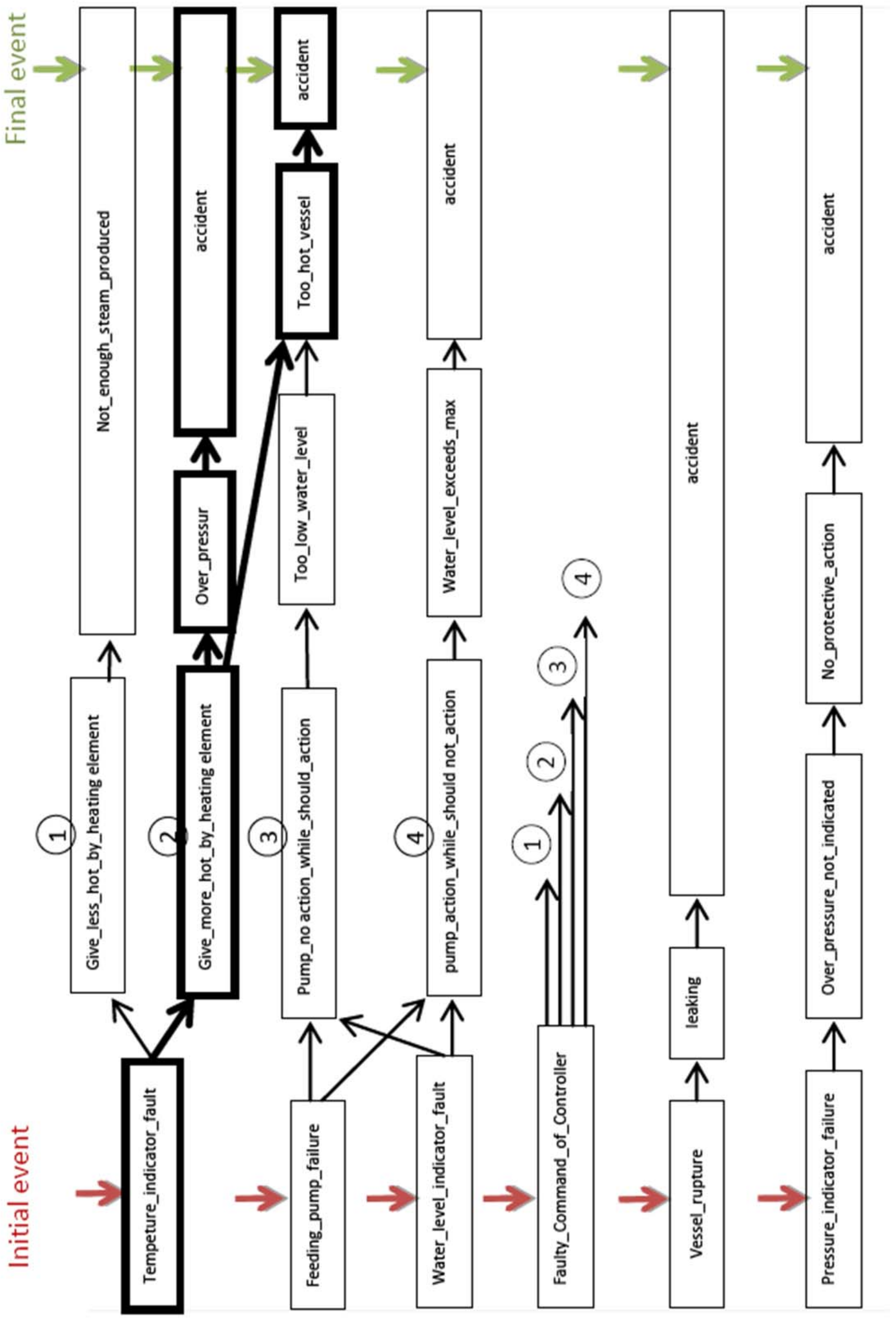
This appendix contains some tests on the developed system model with the steam boiler example. The purpose of the tests is to present that the system is able to generate relevant event chains based on the cause-consequence concepts that are added to the ontology. The following presents some event sequences for the steam boiler system that are created manually using the cause-consequence concepts of the steam boiler system. These cause-consequence concepts were added to the steam boiler ontology.

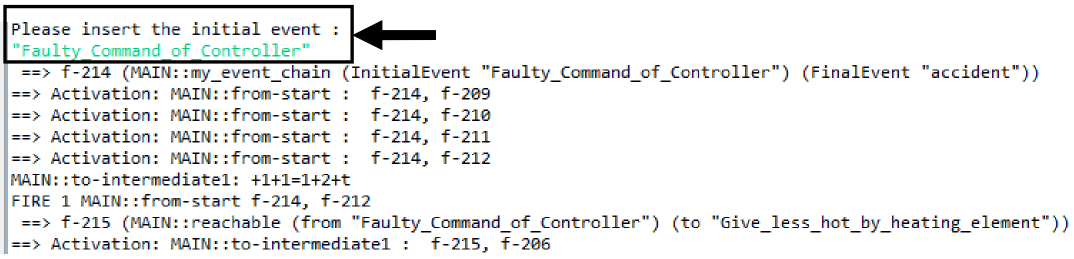

Initial event
Final event
Tempeture_indicator_fault
Give_less_hot_by_heating element
Give_more_hot_by_heating element
Not_enough_steam_produced
Over_pressure
accident
Feeding_pump_failure
Pump_no action_while_should_action
Too_low_water_level
Too_hot_vessel
accident
Water_level_indicator_fault
pump_action_while_should not_action
Water_level_exceeds_max
accident
Faulty_Command_of_Controller
1
2
3
4
Vessel_rupture
leaking
accident
Pressure_indicator_failure
Over_pressure_not_indicated
No_protective_action
accident

After running the program, the user must insert the initial event. Then forward chaining performs over the cause-consequence concepts and expected chains will be created. The chains that do not end with an accident should not be generated by the system.

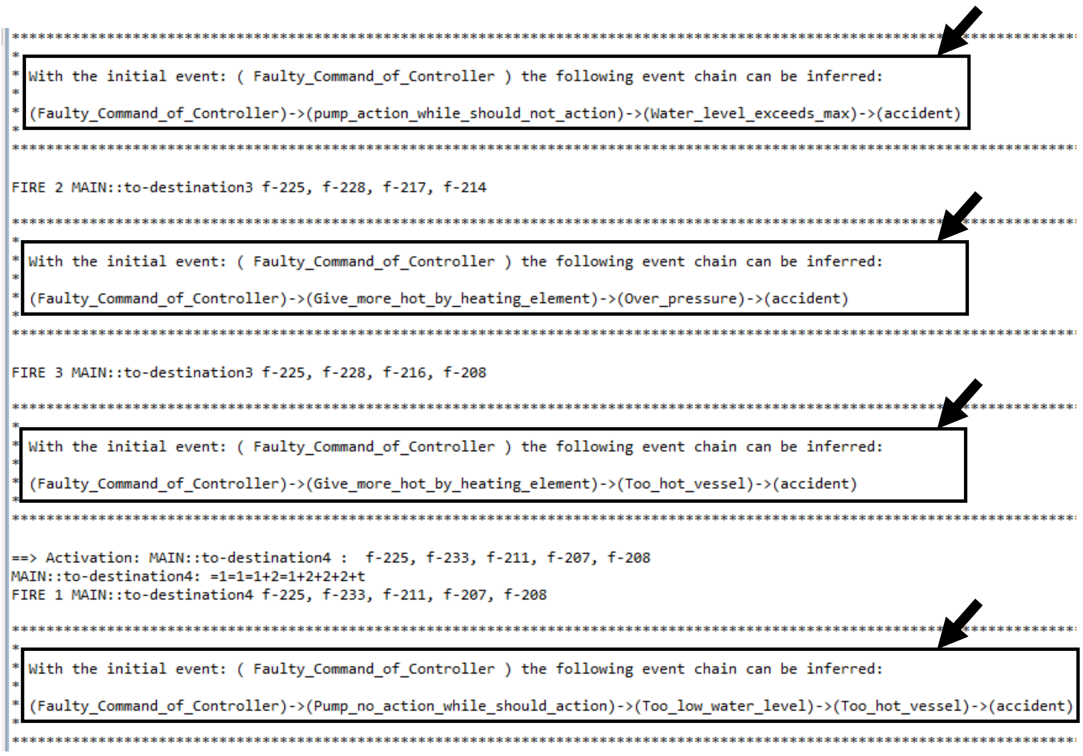


```
Please insert the initial event :
"Tempeture_indicator_fault"
 ==> f-214 (MAIN::my_event_chain (InitialEvent "Tempeture_indicator_fault") (FinalEvent "accident")
==> Activation: MAIN::from-start :  f-214, f-197
==> Activation: MAIN::from-start :  f-214, f-198
MAIN::to-intermediate1: +1+1=1+2+t
FIRE 1 MAIN::from-start f-214, f-198
 ==> f-215 (MAIN::reachable (from "Tempeture_indicator_fault") (to "Give_less_hot_by_heating_elemen
==> Activation: MAIN::to-intermediate1 :  f-215, f-206
FIRE 2 MAIN::to-intermediate1 f-215, f-206
 ==> f-216 (MAIN::reachable (from "Give_less_hot_by_heating_element") (to "Not_enough_steam_produce
FIRE 3 MAIN::from-start f-214, f-197
 ==> f-217 (MAIN::reachable (from "Tempeture_indicator_fault") (to "Give_more_hot_by_heating_elemen
==> Activation: MAIN::to-intermediate1 :  f-217, f-204
==> Activation: MAIN::to-intermediate1 :  f-217, f-205
FIRE 4 MAIN::to-intermediate1 f-217, f-205
 ==> f-218 (MAIN::reachable (from "Give_more_hot_by_heating_element") (to "Over_pressure"))
==> Activation: MAIN::to-intermediate1 :  f-218, f-202
FIRE 5 MAIN::to-intermediate1 f-218, f-202
 ==> f-219 (MAIN::reachable (from "Over_pressure") (to "accident"))
FIRE 6 MAIN::to-intermediate1 f-217, f-204
 ==> f-220 (MAIN::reachable (from "Give_more_hot_by_heating_element") (to "Too_hot_vessel"))
==> Activation: MAIN::to-intermediate1 :  f-220, f-196
FIRE 7 MAIN::to-intermediate1 f-220, f-196
 ==> f-221 (MAIN::reachable (from "Too_hot_vessel") (to "accident"))
MAIN::to-destination1: =1=1=1+2+t
MAIN::to-destination2: =1=1=1+2=1+2+t
==> Activation: MAIN::to-destination3 :  f-214, f-217, f-204, f-196
==> Activation: MAIN::to-destination3 :  f-214, f-217, f-205, f-202
MAIN::to-destination3: =1=1=1+2=1+2+2+t
FIRE 1 MAIN::to-destination3 f-214, f-217, f-205, f-202

*************************************************************************************************
*
* With the initial event: ( Tempeture_indicator_fault ) the following event chain can be inferred:
*
* (Tempeture_indicator_fault)->(Give_more_hot_by_heating_element)->(Over_pressure)->(accident)
*
*************************************************************************************************

FIRE 2 MAIN::to-destination3 f-225, f-228, f-216, f-208

*************************************************************************************************
*
* With the initial event: ( Tempeture_indicator_fault ) the following event chain can be inferred:
*
* (Tempeture_indicator_fault)->(Give_more_hot_by_heating_element)->(Too_hot_vessel)->(accident)
*
*************************************************************************************************
```

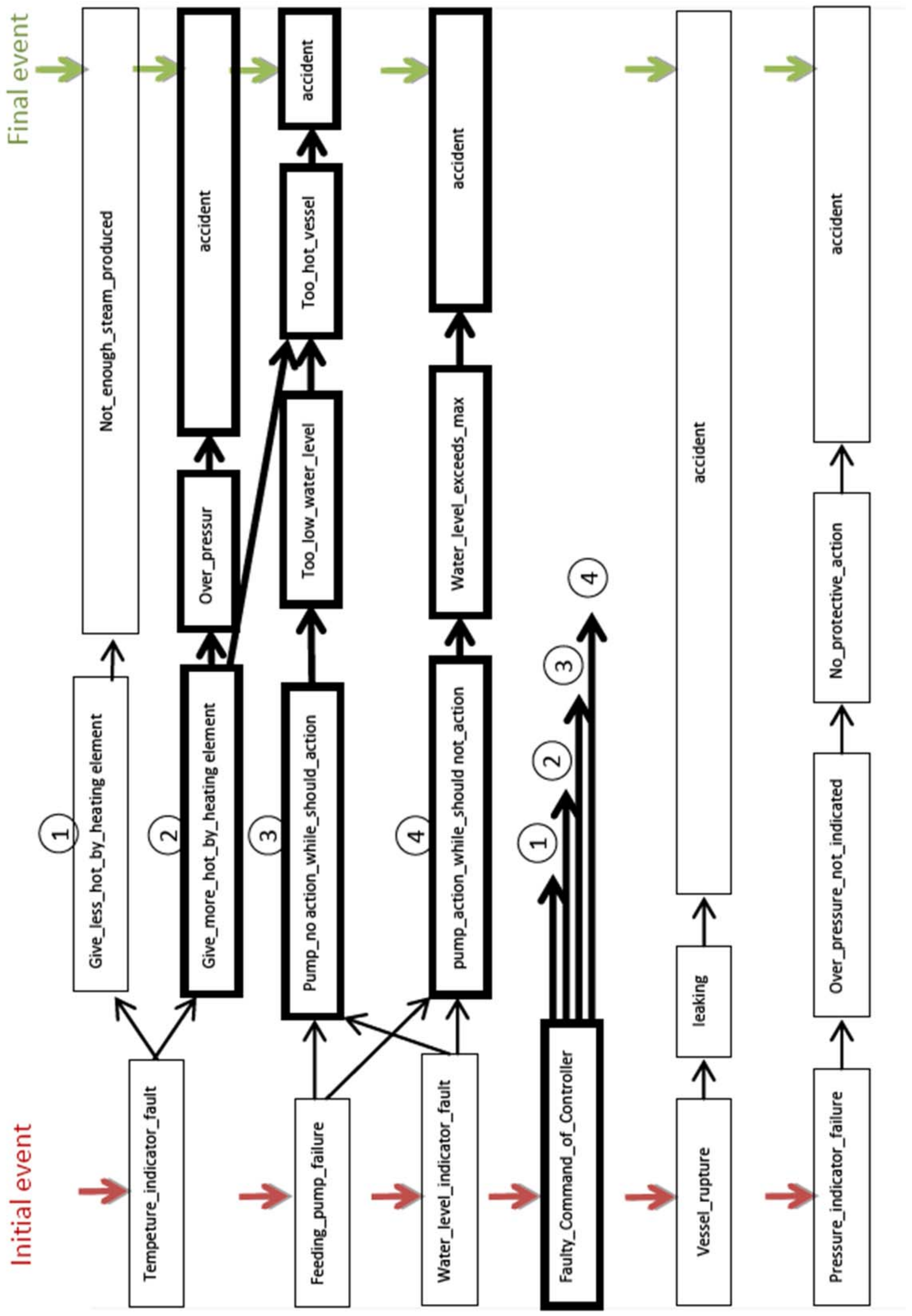

Initial event
Final event
Tempeture_indicator_fault
Give_less_hot_by_heating element
Not_enough_steam_produced
Give_more_hot_by_heating element
Over_pressur
accident
Too_hot_vessel
accident
Feeding_pump_failure
Pump_no action_while_should_action
Too_low_water_level
Water_level_indicator_fault
pump_action_while_should not_action
Water_level_exceeds_max
accident
Faulty_Command_of_Controller
1
2
3
4
Vessel_rupture
leaking
accident
Pressure_indicator_failure
Over_pressure_not_indicated
No_protective_action
accident

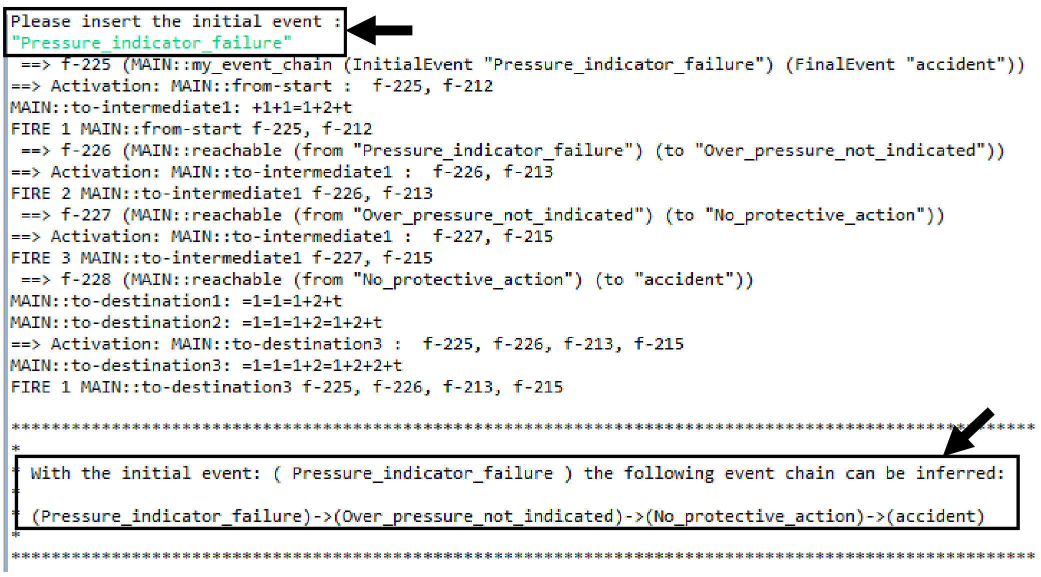


```
Please insert the initial event :
"Faulty_Command_of_Controller"
 ==> f-214 (MAIN::my_event_chain (InitialEvent "Faulty_Command_of_Controller") (FinalEvent "accident"))
==> Activation: MAIN::from-start :  f-214, f-209
==> Activation: MAIN::from-start :  f-214, f-210
==> Activation: MAIN::from-start :  f-214, f-211
==> Activation: MAIN::from-start :  f-214, f-212
MAIN::to-intermediate1: +1+1=1+2+t
FIRE 1 MAIN::from-start f-214, f-212
 ==> f-215 (MAIN::reachable (from "Faulty_Command_of_Controller") (to "Give_less_hot_by_heating_element"))
==> Activation: MAIN::to-intermediate1 :  f-215, f-206
```

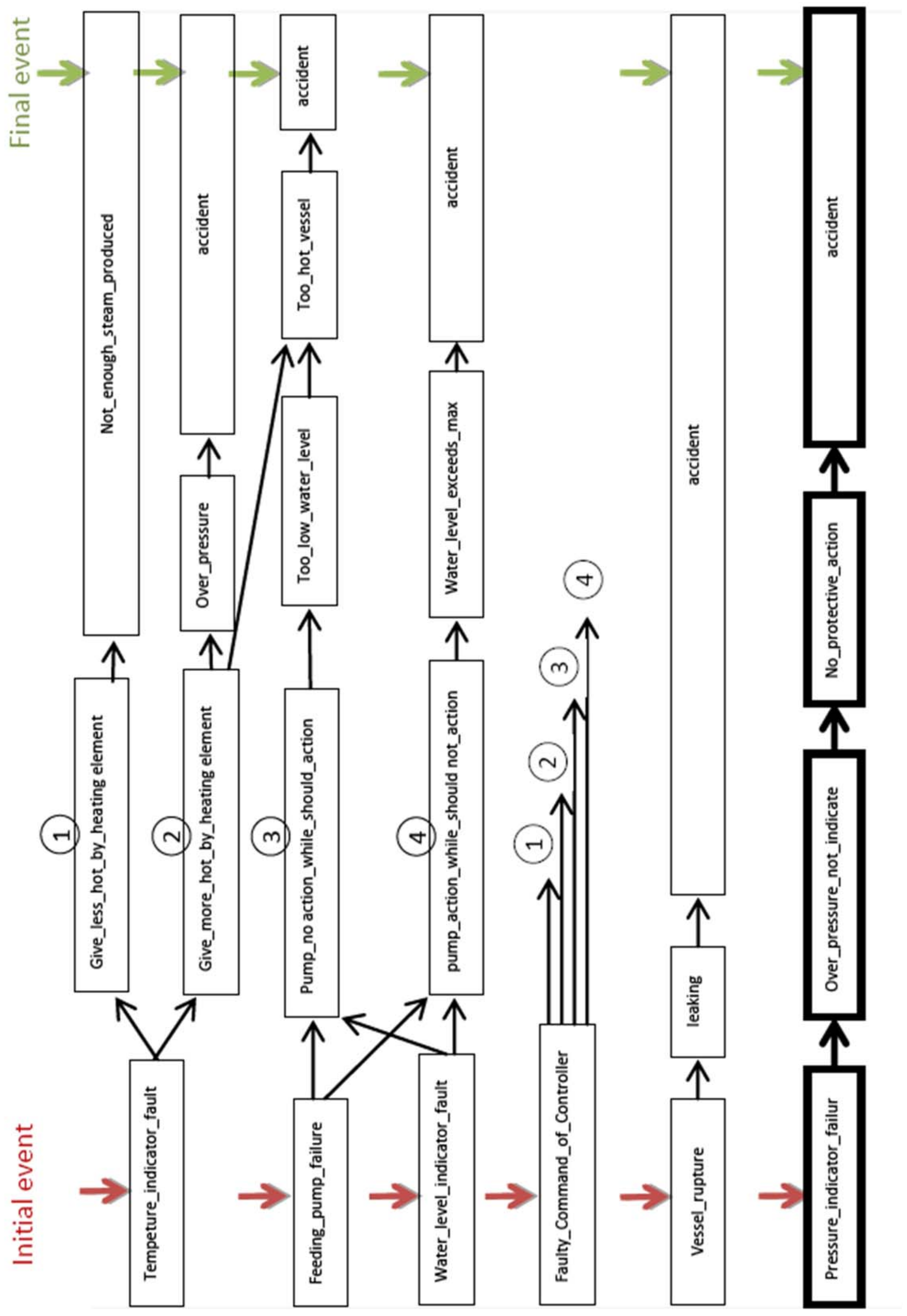


```
**********************************************************************************************************
*
* With the initial event: ( Faulty_Command_of_Controller ) the following event chain can be inferred:
*
* (Faulty_Command_of_Controller)->(pump_action_while_should_not_action)->(Water_level_exceeds_max)->(accident)
*
**********************************************************************************************************

FIRE 2 MAIN::to-destination3 f-225, f-228, f-217, f-214

**********************************************************************************************************
*
* With the initial event: ( Faulty_Command_of_Controller ) the following event chain can be inferred:
*
* (Faulty_Command_of_Controller)->(Give_more_hot_by_heating_element)->(Over_pressure)->(accident)
*
**********************************************************************************************************

FIRE 3 MAIN::to-destination3 f-225, f-228, f-216, f-208

**********************************************************************************************************
*
* With the initial event: ( Faulty_Command_of_Controller ) the following event chain can be inferred:
*
* (Faulty_Command_of_Controller)->(Give_more_hot_by_heating_element)->(Too_hot_vessel)->(accident)
*
**********************************************************************************************************

==> Activation: MAIN::to-destination4 :  f-225, f-233, f-211, f-207, f-208
MAIN::to-destination4: =1=1=1+2=1+2+2+2+t
FIRE 1 MAIN::to-destination4 f-225, f-233, f-211, f-207, f-208

**********************************************************************************************************
*
* With the initial event: ( Faulty_Command_of_Controller ) the following event chain can be inferred:
*
* (Faulty_Command_of_Controller)->(Pump_no_action_while_should_action)->(Too_low_water_level)->(Too_hot_vessel)->(accident)
*
**********************************************************************************************************
```


Initial event
Final event
Tempeture_indicator_fault
Give_less_hot_by_heating element
Not_enough_steam_produced
Give_more_hot_by_heating element
Over_pressur
accident
Feeding_pump_failure
Pump_no action_while_should_action
Too_low_water_level
Too_hot_vessel
accident
Water_level_indicator_fault
pump_action_while_should not_action
Water_level_exceeds_max
accident
Faulty_Command_of_Controller
1
2
3
4
Vessel_rupture
leaking
accident
Pressure_indicator_failure
Over_pressure_not_indicated
No_protective_action
accident



```
Please insert the initial event :
"Pressure_indicator_failure"
 ==> f-225 (MAIN::my_event_chain (InitialEvent "Pressure_indicator_failure") (FinalEvent "accident"))
==> Activation: MAIN::from-start :  f-225, f-212
MAIN::to-intermediate1: +1+1=1+2+t
FIRE 1 MAIN::from-start f-225, f-212
 ==> f-226 (MAIN::reachable (from "Pressure_indicator_failure") (to "Over_pressure_not_indicated"))
==> Activation: MAIN::to-intermediate1 :  f-226, f-213
FIRE 2 MAIN::to-intermediate1 f-226, f-213
 ==> f-227 (MAIN::reachable (from "Over_pressure_not_indicated") (to "No_protective_action"))
==> Activation: MAIN::to-intermediate1 :  f-227, f-215
FIRE 3 MAIN::to-intermediate1 f-227, f-215
 ==> f-228 (MAIN::reachable (from "No_protective_action") (to "accident"))
MAIN::to-destination1: =1=1=1+2+t
MAIN::to-destination2: =1=1=1+2=1+2+t
==> Activation: MAIN::to-destination3 :  f-225, f-226, f-213, f-215
MAIN::to-destination3: =1=1=1+2=1+2+2+t
FIRE 1 MAIN::to-destination3 f-225, f-226, f-213, f-215

*************************************************************************************************
*
* With the initial event: ( Pressure_indicator_failure ) the following event chain can be inferred:
*
* (Pressure_indicator_failure)->(Over_pressure_not_indicated)->(No_protective_action)->(accident)
*
*************************************************************************************************
```



Initial event
Final event
Tempeture_indicator_fault
Give_less_hot_by_heating element
Not_enough_steam_produced
Give_more_hot_by_heating element
Over_pressure
accident
Feeding_pump_failure
Pump_no action_while_should_action
Too_low_water_level
Too_hot_vessel
accident
Water_level_indicator_fault
pump_action_while_should not_action
Water_level_exceeds_max
accident
Faulty_Command_of_Controller
1
2
3
4
Vessel_rupture
leaking
accident
Pressure_indicator_failur
Over_pressure_not_indicate
No_protective_action
accident