

Automated Ontology Evolution: Utilising Bridging Rules

Punyanuch Borwarnginn



Master of Science
School of Informatics
University of Edinburgh
2011

Abstract

The thesis extends McNeill's Ontology Repair System (ORS). ORS is an automated system for ontology repair which demonstrate the problem of ontological mismatches between agents in a multiagent system. In the current ORS, the agent in ORS keeps only one copy of the ontology is kept. When mismatches are identified and repaired, the changes are made to this ontology, with no information about the previous repair being kept. This approach risks the agents losing the useful information, making it unable to interact with agents that it previously had successful interactions with. In this project, we extended ORS to enable it to keep and reuse the repair information. We introduce the term '*bridge rule*', which is the rule extracted from the repair information. We hypothesize that keeping bridge rules leads to fewer failures and fewer repairs compared to the original ORS. We present the design structure of our bridge rules and our implementation of bridge rules as one of component in ORS. The evaluation of the system confirms our hypothesis and demonstrates the effectively and efficiency performance.

Acknowledgements

I would like to thank my supervisor, Fiona McNeill. Without her support, guidance, and useful comments, the project could be finished. She spent times to read my drafts and always gave useful feedbacks. I am also thankful to my second supervisor, Liwei Deng, for his helpful comments.

In addition, I would like to thank my family for always supporting and encouraging me throughout this year.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Punyanuch Borwarnginn)

Table of Contents

Chapter 1	Introduction.....	1
Chapter 2	Background.....	3
2.1	Agents in Multiagent systems	3
2.2	Ontologies and their representations	4
2.3	Ontology Matching	6
2.4	Ontology Versioning.....	7
2.5	Bridge Rules.....	8
Chapter 3	Ontology Repair System	10
3.1	System overview	10
3.2	Subsystems of Ontology Repair System	12
3.2.1	Plan Deconstructor.....	12
3.2.2	Diagnostic system.....	12
3.2.3	Repair system.....	12
3.2.4	Translation System.....	12
3.3	Diagnosis and Repair	13
3.3.1	Types of Mismatches	13
3.3.2	Diagnostic Algorithm.....	15
3.3.3	Repair.....	17
Chapter 4	Design and Implementation	18

4.1	System Overview	18
4.2	Bridge Rules	20
4.3	Storing Bridge Rules	22
4.3.1	ORS Repair Information	23
4.3.2	Bridge Rules.....	26
4.4	Searching and Consuming Bridge Rules.....	31
Chapter 5	Evaluation	35
5.1	Effectiveness and Efficiency	35
5.2	The Shopping Ontology	37
5.3	The Experiment	37
5.3.1	Ontologies	38
5.3.2	Bridge Rules.....	40
5.3.3	Test cases for evaluations	41
5.4	Results	42
5.4.1	Raw results.....	43
5.4.2	Line graph	44
5.5	Analysis.....	49
5.5.1	Evaluation on Effectiveness.....	49
5.5.2	Evaluation on Efficiency.....	50
5.6	Related Works	51
Chapter 6	Further Works	53
Chapter 7	Conclusion	55
Bibliography		57
Appendix A	The Shopping Ontology	60
A.1	Shopping A.....	60
A.2	Shopping B	62

A.3	Shopping C.....	64
Appendix B The Stored Bridge Rules		67
B.1	Set A.....	67
B.2	Set B	67
B.3	Set C	67
Appendix C The Output for the main test cases		69
C.1	The Output from Case 1	69
C.2	The Output from Case 5	71
C.3	The Output from Case 9	76

List of Figures

Figure 3.1: An overview of ORS interacting with a planning agent.	11
Figure 4.1: The ontology repair system flow chart with utilising bridge rules.....	20
Figure 4.2: The structure of a bridge rule.	21
Figure 4.3: The overview of searching bridge rule method.....	34
Figure 5.1 : The different mismatches between three shopping ontologies.	39
Figure 5.2: The nine test cases from a combination of each ontology and bridge rule.	42
Figure 5.3: The three test cases without sets of bridge rules.	42
Figure 5.4: The line graph of duration time for running the test cases involved Shopping A.	45
Figure 5.5: The cumulative time of running the test cases involved Shopping A.	45
Figure 5.6: The line graph of duration time for running the test cases involved Shopping B.....	46
Figure 5.7: The cumulative time of running the test cases involved Shopping B.	46
Figure 5.8: The line graph of duration time for running Shopping A to see how much time the ORS waste with bridge rule component.	47
Figure 5.9: The cumulative time of running Shopping A to see how much time the ORS waste with the bridge rule component.....	47
Figure 5.10: The line graph of duration time for running Shopping B to see how much time the ORS waste with bridge rule component.....	48
Figure 5.11: The cumulative time of running Shopping B to see how much time the ORS waste with the bridge rule component.....	48

List of Tables

Table 5.1: The tables of running times in second for each test case.....	43
Table 5.2: The table of the number of the internal action failed and performing repair before achieving the goal.	43

Chapter 1

Introduction

Ontologies are logical descriptions for representing knowledge in generally or in a specific domain. Their major role is representing knowledge including logic and vocabulary for an agent. In the traditional approach, ontologies have been manually built for a specific application. Recently, the growth of Semantic Web has increased the uses of ontologies for communicating between agents. If two agents want to communicate at least their ontologies must match on that action, which means it is not necessary that their ontologies fully align but it should align in the part that is essential for communication. However, it is generally impossible that agents' ontologies always align because they are not typically made from a single person. Therefore, some of them might need to change their ontologies to the failure communication.

McNeill's Ontology Repair System (ORS) is an automated system for ontology repair. It diagnoses and evolves ontology of a planning agent (PA) to match with service-providing agents (SPAs) to achieve goals even if failure occurs. Currently, ORS only keeps one version of the PA's ontology after repair. The changes are made to this ontology so no previous information is kept. If we consider that the agent may need previous ontologies to communicate with other or previous agents, this approach risks the agents losing the useful information for interacting with the previous agents and their common representations that have been altered by ORS. In this project, we introduce the new component that can manage to store and use the repair information from ORS. We call this stored information as '*bridge rules*'. Since we believe that keeping the bridge rules leads to less repair and decreases the

execution failures, we then investigate and implement an extension of ORS to test our belief in the project. This thesis is organised as follows:

Chapter 2 provides the background of this project, including AI techniques and terms that related to ORS such as agents, ontologies and relevant works on ontology versioning and bridging rules.

Chapter 3 provides an overview of ORS, describing each sub-system and the system flow. Since this project is an extension of ORS, the chapter is to ensure that the reader is familiar with the concept of ORS before continue to our work.

Chapter 4 presents our design and implementation of an extended component of ORS. It describes how we structure a bridge rule and use it in the ORS.

Chapter 5 provides an evaluation of the system on how effective and efficient it is compared to the original ORS.

Chapter 6 discusses further works in which this project can be expanded.

Chapter 7 gives the conclusion of the thesis.

Chapter 2

Background

The theory of ORS relates to many areas in Artificial Intelligence such as agents and ontologies. Before introducing ORS in chapter 3, we provide an overview of the related works done in the fields on which the project depends. This is to ensure the reader has basic concepts and understandings of technologies and terms which will appear later in the project. This covers some relevant works on agents in multiagent systems, ontologies, ontology matching and versioning.

2.1 Agents in Multiagent systems

In Artificial Intelligence, an agent can be considered as “*a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives*” [26]. Intelligent agents are generally expected to include the following characteristics [23, 26, 27]:

- **Autonomy:** agents should initiate actions without external intervention and reactivity refers to agents that are able to perceive and respond to their environment.
- **Reactivity:** agents should perceive and respond in suitable ways to changes in their environment.
- **Pro-activeness:** agents should be able to exhibit goal-directed behaviour by taking the initiative.
- **Social ability:** agents should interact with other agents and possibly humans to achieve their objectives.

Our main focus is multiagent systems. Multiagent systems are generally distributed systems that allow interact between several independent agents to perform tasks and reach their goals which cannot be achieved by only the individual agent. In short, they are systems composed of multiple interacting agents [5]. However, when the communication involves many individual agents, there are some difficulties for agents to work together because agents are generally implemented by different people. These problems are called distributed problems. In order to solve these problems, they require that agents need coherence (i.e., agents need to want to work together) and competence (i.e., agents need to know how to work well together) [4].

If agents are implemented at different time using different language, rules or ontologies, the problem of competence can be rather complex in the real world because it is difficult to know how to communicate with others. The aim of ORS could be related to this problem that is to reduce the problem of competence where agents do not share the same ontology.

Therefore, we can consider that this project attempts to improve ORS by storing different versions of the ontology, in order to make a fewer ontology repairs in multiagent systems and to avoid losing information.

2.2 Ontologies and their representations

The term “*ontology*” can be described in many different ways depending on the area of studies. Ontology is originated from philosophy. It referred to the study of what exist along with their structures, properties and relations in reality [25] and try to describe the existing subject in the real world. In general, ontology is used in Artificial Intelligence to refer to the knowledge representation of a system. In this section, we briefly introduce its definitions and give clarification on what we mean by ontology in this project.

According to Gruber [9, 10], an ontology is defined as “*an explicit specification of a conceptualisation*”. A conceptualisation is “*an abstract, simplified view of the world that is represented in some purpose*” [9]. It represents the formal definitions and concepts that could be shared in the knowledge-based environment. In short, an ontology can be defined as “*a conceptual model*” [25] that describes the objects,

concepts and relationships between them in some specific domains. In Kalfoglou's survey article [14], he explored several ontologies definitions and summarized it as "*an explicit representation of a shared understanding of the important concepts in some domain of interest*". Furthermore, an ontology is considered as "*a set of logical axioms designed to account for the intended meaning of a vocabulary*" [11] indicating that ontologies are logic based. In general, the major roles of ontologies are as knowledge representation within the agent itself and to support knowledge sharing and reuse with others.

Many ontologies are represented as taxonomic hierarchies of classes that describe the subtype relationships such as *is-a* and *part-of* between them. However, the taxonomy only contains the terminologies and simple relationships instead of the complex relationships and facts that describe the knowledge of the world.

In addition to taxonomy, there are several ways to represent ontology. An ontology can be represented by the web ontology language (OWL). OWL is introduced and recommended by W3C for representing ontology in the Semantics Web environment [2]. It extends the format of Resource Description Resource (RDF) and RDF Schema (RDFS). OWL can extend the class hierarchy relationships by supporting the Description Logics (DL). Description Logics describes concepts, roles, and individuals by using logic-based semantics syntax (i.e., first-order logic) [1].

Knowledge Interchange Format (KIF) is another popular ontology representation format. KIF is "*a formal language for the interchange of knowledge among disparate computer programs*" [7]. This computer programs are written by different programmers at different times and different languages. KIF is based on the first order logic notation. Furthermore, Ontolingua Server was introduced as a system for analyzing and translating ontologies [8]. Ontologies written on this server are written in representation which is based on KIF [6].

In this project, we use ontologies which are compatible with ORS. Ontologies are written in KIF. Ontologies were divided into two parts: the signature and the theory [16]. The signature describes the representation language that represents such as the type hierarchy, the names and the types of predicates and functions. The theory is the definitions, axioms and some rules that are defined by the signature.

2.3 Ontology Matching

We described an ontology the purpose as knowledge representation and knowledge sharing about the world. However, ontologies are created from a different group of people which different interest. They have the different points of view to represent ontologies. For example, one talks about a car but the other talks about a vehicle. As humans, we can notice that a vehicle is a superclass of a car, but machine cannot intuitive understand this meaning unless it is explicitly stated in an ontology. Since we need to use ontologies to communicate between agents in the multi-agent system, ontology matching approaches are considered.

There are many techniques that are developed for matching between ontologies, the following techniques were discussed in [15]:

- **Ontology mapping**

According to Kalfoglou and Schorlemmer, ontology mapping defines as “*The task of relating the vocabulary of two ontologies that share the same domain of discourse in such a way that the mathematical structure of ontological signatures and their intended interpretations, as specified by the ontological axioms, are respected*” [15]. The general idea is to define a map between two or more ontologies that describe the same domain and sometimes using tools to alter them.

- **Ontology merging**

Unlike ontology mapping, ontology merging creates a new single ontology result from merging two or more existing ontologies and contain information that cover from both of them. Merging is defined as “*the act of building a new ontology by unifying several ontologies into a single one*” [12] and “*a single coherent ontology that includes all information from all the sources*” [19].

- **Ontology alignment**

Alignment is a process in which “*the sources must be made consistent and coherent with one another but kept separately*” [19] that involves ontology mapping and defines relationships between them to indicate the consistency.

- **Ontology translation**

Translation can be considered as an implementation of the ontology mapping. After mapping defines a collection of functions about concepts and relations that relate to others. Ontology translation is the application of the mapping functions to translate the sentences based on one ontology into the other [15].

However, these techniques generally require full access to ontologies and are done in offline mode. These requirements make it is difficult in practice especially in the Semantic Web. In the real world, not all ontologies are open to access as some of them are commercial or contain some private information such as their business processes and flow. They only allow some part of ontologies to be shared for making communication successfully. Moreover, the approaches are offline systems, which are not dynamic in the multi-agent system. In the Semantic Web context, the interaction between different agents happens most of the time which ontological matches may be occurs during these executions. Therefore, the offline ontology matching approaches might not be able to capture all mismatches for this situation. However, the solution for the problems has been explored in ORS which will be explained in chapter 3.

2.4 Ontology Versioning

As described in the section 2.3 about ontology matching, there are some questions about how to keep each version of the ontology after perform the matching process. In this section, we describe some existing works about versioning ontology. However, most of them are different from what we are trying to work in this project as will be discussed in Chapter 4.

One of the relevant works is proposed in [21, 22]. They presented a framework for tracking changes called a change detection approach as the component of their ontology evolution for OWL DL ontologies within a closed environment. The approach introduced three main components: the *version log*, the *change definition language* (CDL) and *evolution log*. The version log is only used for keeping track of the different version for all concepts that is created and modified within the ontology. However, it does not specify what is changed in the version. To identify what has

changed, the CDL is used to specify them. CDL is a language based on temporal logic that expresses the change in a formal and declarative way [22]. The collection of change definitions that conform to the version log is added into the evolution log that keep track on all the changes history of the ontology.

Another approach introduced in [20] is called *PromptDiff* which is the component in the PROMPT ontology management framework [19] with based on the frame structure. This framework helps the user to merging ontologies along with identifying the different version in the same ontology. There are three main components: *iPrompt* as an interactive ontology-merging, *AchorPrompt* as a graphical based tools for ontology mapping, and *PromptDiff* as an ontology-versioning tool. We focus on *PromptDiff* versioning tools. *PromptDiff* approach is to compare the structures in different ontology versions. For instance, *PromptDiff* take two different ontology versions as input it will automatically create a table that show the differences between their structures that what has been changed within each frame and hierarchy.

In both approach, they were implemented as plugins or extensions to the Protégé ontology development environment. Thus they are required the full accessed to an ontology and perform in off-line mode, whereas ORS is only able to access some parts of an ontology which are naturally released during agent communication and perform during the run time. However, these two approaches could be useful and effective after the execution.

2.5 Bridge Rules

As we discussed in the existing works on keeping the versions of an ontology, these are effective mostly in off-line mode. However, our main focus is keeping this information during the execution time, and it needs to be compatible with ORS. ORS diagnoses and repairs the ontology without keeping any repair information, the details will be discussed in chapter 3. In this project, we attempt in keep this repair information which we call ‘bridge rules’. In this section, we provide some definitions and existing works related to bridge rules before introducing our work in Chapter 4.

In general, a bridge rule is the result from ontology mapping process. In [3, 18], “*a mapping between two ontologies is a set of bridge rules*”. Each bridge rule in the mapping connects a concept from one ontology to a concept from another. It expresses the semantic connection between these two ontologies [24]. A bridge rule generally has two parts: *into-bridge rule* and *an onto-bridge rule* [18, 24]. The into-bridge rule has a form: $i:C \xrightarrow{\sqsubseteq} j:D$, which means the concept C in i-th ontology is less general than or as general as the concept D in j-th ontology. Whereas the onto-bridge rule has a form: $i:C \xrightarrow{\sqsupseteq} j:D$, which means in contrast to the onto that the first term is more general than the second term. In addition to these two expressions, there are three more operators as explained in [3]: \equiv , \perp and $*$. \equiv means that two concepts are the same level of abstraction. \perp states that two concepts are completely disjoint or unrelated. Finally, $*$ expresses that two concepts are compatible and may be related.

There are several works involving bridge rules. Mostly bridge rules are used for checking ontology inconsistency. [3] investigates the problem of how to maintain the local and shared ontology to be consistent after ontology mapping is performed. It introduced a language Context OWL (C-OWL) that extended the OWL syntax by adding bridge rules which allow to related information in different ontologies at the semantic level; the details are given in [3]. Another work is done in [18] which addresses the problem of errors in the ontology mapping by proposing an automatic system for debugging ontology mappings. Bridge rules are used to find inconsistencies by checking whether any bridge rules conflicts after several mappings perform called “*logical inconsistencies*”, then it discovers and repairs these inconsistencies by removing bridge rules that conflict with the ontology and undoing correspond ontology mappings [18].

Chapter 3

Ontology Repair System

McNeill's Ontology Repair System (ORS) is a system designed to solve the problem of ontological mismatches, which lead to communication problems between agents during run-time, without involving human interaction [16]. It aims to make the communication become possible in a large scale system (i.e., the Semantic Web [2]). In the project, we can consider it as an extension module for ORS that we will focus on keeping the repair information from ORS. In this chapter, we introduce an overview of ORS in order to ensure that the reader understands and is familiar with key concepts and a system overview before we discuss our design in chapter 4.

3.1 System overview

ORS is designed as a plug-in for any agent called planning agent (PA) that assumes problem occurs with the PA. The PA tries to achieve its goals by planning actions and communicating with other agent called Service Providing Agent (SPA) to perform these actions. If the execution fails, ORS diagnoses the problems. ORS automatically repairs the PA's ontology if it causes the failure. The main role is to help the PA to fix its ontology during the run time then it can continue their interaction.

ORS consists of four modules [16, 17]: *plan deconstructor*, *diagnosis*, *repair and translation*. When the execution plan fails, the *plan deconstructor* deconstructs the plan with reference to the ontology that generated the plan: each action is annotated with the action rule that describes it and a justification for each precondition as to why it is true. The *diagnostic module* will diagnose the problem and identify the

types of mismatch. In the *repair module*, the PA's ontology is repaired to make it align with SPA's ontology. There is also a *translation module* which creates a working ontology in Prolog from the KIF ontology, because both the PA and ORS are written in Prolog. The overview is illustrated in Figure 3.1.

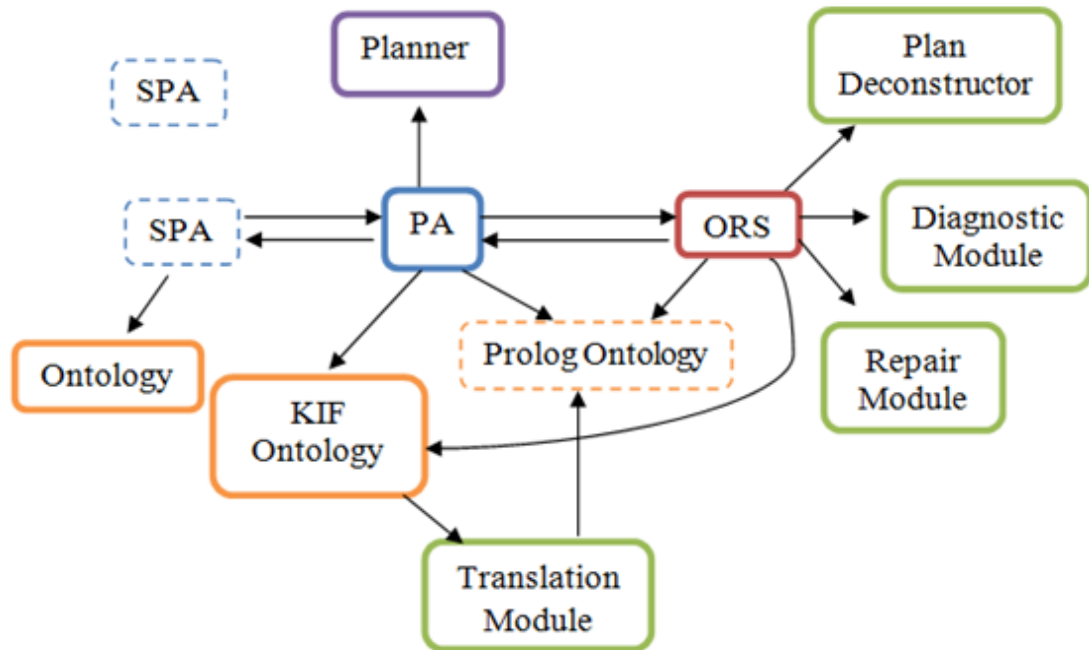


Figure 3.1: An overview of ORS interacting with a planning agent.

When the PA receives the goal, it calls translation module to translate the KIF ontology into Prolog ontology. Then the PA carries out the action plans that correspond to the goal. When failure occurs, i.e., SPAs fail to execute the plan which in the case, we first assume to be an ontological mismatch, ORS is called after this action. . ORS *deconstructs* the plan by discussing the justification for the original plan with its ontology. The diagnostic module uses the information from the plan deconstructor and performs the diagnostic algorithm to identify what went wrong. If a fault is diagnosed and can be repaired, the diagnostic module passes the information to the repair module which will repair the KIF ontology. After the repair successful, the PA restarts the whole action again with the updated ontology and attempts to accomplish the goal. The process terminates either the goal has been reached, or the system could not compose the plan with the updated ontology.

3.2 Subsystems of Ontology Repair System

As we discuss in section 3.1, ORS consists of four subsystems. In the section, we briefly discuss these subsystems which are also fully explained in [16, 17].

3.2.1 Plan Deconstructor

When the plan has failed, this plan will be deconstructed by the plan deconstructor to produce a first-order ontological justification for the original plan produced by the planner. The plan deconstructor justifies the plan by finding the relations to the ontology. We can think of the action in the plans are equal to rules in ontology and preconditions are also in the actions in the plan. The justified plan is a list that contains relevant information, e.g., the action name, the identifier for the rule and the preconditions for the action. This information is passed through the diagnostic system for identifying the problem.

3.2.2 Diagnostic system

This system is the major part of ORS. When the SPA fails or refuses to carry out the action, this failure could be caused by the ontological mismatch. The diagnostic system receives the information either from PA itself or the justified plan. It attempts to diagnose the problem and identify a type of mismatches which will be discussed later in section 3.3.

3.2.3 Repair system

This system is immediately called after the diagnostic system. The diagnostic system identifies the mismatch and sends this information to the repair system. If the mismatch can be repaired, the repair system will repair it by altering the PA's ontology. Then the communication can be restarted again and attempt to reach the goal.

3.2.4 Translation System

Translation system was implemented because the system is working in two different representations which are KIF and Prolog. The ontologies were written in KIF. However, the agents and ORS were written in Prolog and can most easily work with prolog ontologies. Therefore, the translation between KIF and Prolog is required.

Before any action starts, the KIF ontology is translated to the Prolog ontology. During the execution, any ontological changes are made in Prolog ontology, but when the plan terminates or fails, the KIF ontology is updated with all these changes.

3.3 Diagnosis and Repair

In this section, we mainly focus on the heart of ORS which are diagnosis and repair ontology. Diagnosis identifies the type of mismatch between the PA and SPAs. Ontologies are typically mismatched because they represent information in different level so that either the PA or SPAs could not understand their meaning. For instance, if the goal is to rent a car and the PA plan to communication with a car agent. However, the plan fails during execution, then ORS diagnoses that the PA has an ontological mismatch that is itself asking for a *vehicle(X)* but SPA replies a *car(X)* which we know that car is subtype of vehicle. This mismatch type is called *refinement*. Refinement means to specialise the concepts in more detail as we see in the example the car is more specific than vehicle. In contrast, if the PA ontology contains *car(X)* and SPA represents it as *vehicle(X)*. This mismatch is called *abstraction*, which attempts to generalise the concepts and this can be repaired by removing detail. In this section, we will briefly discuss types of mismatches, the diagnostic algorithm and repair. The full details can be found in [16, 17].

3.3.1 Types of Mismatches

In order to perform diagnosis and repair the ontology, we need to understand the characteristics of ontological mismatches and can identify where and what differ to others. In this section, we introduce the types of mismatches that relevant to the scope of the project.

Abstraction

This is when our ontology contains too much information and it represents concepts and domains in too specific ways. This creates the need to alter the ontology to be more general by removing detail. This technique is called performing an *abstraction* of the ontology. The following are four main categories that abstraction can occur:

1. Predicate Abstraction

The process of matching the predicate then the predicate name is mapped into more general way,

e.g., $Car(X)$, $Bicycle(X)$ map to $Vehicle(X)$

2. Domain Abstraction

The process of matching the class of arguments, constants, and functions then they are mapped into more general way. It can be done by moving up the class level in the class hierarchy,

e.g., $Buy(Agent, AiBook)$, $Buy(Agent, CsBook)$ map to $Buy(Agent, Book)$

3. Propositional Abstraction

The process of changing the number of arguments by dropping some or all of the arguments in a predicates then the arity is less than before,

e.g., $Money(X, Currency, Amount)$ map to $Money(X, Amount)$

4. Precondition Abstraction

The process of changing the preconditions of an action rule, this is done by dropping one or more preconditions,

e.g., $Buy(Agent, Book) \rightarrow Has(Agent, Book)$ map to $Has(Agent, Book)$

Refinement

In contrast to abstraction, when our ontology contains only general concepts and is not detailed enough, *refinement* performs the opposite task by adding detail to the ontology. Then the ontology change to specialise and contains enough information to enable communication with others. The following are four main categories that refinement can occur and equivalent to invert of abstraction:

1. Predicate Refinement

The process of splitting a single predicate into one or more subtype then these updated predicates contain more specific information,

e.g., $Book(X)$ maps to $AcademicBook(X)$, $Novel(X)$

2. Domain Refinement

The process of changing the type of argument by dividing into one or more subtypes,

e.g., $Join(Agent, Group)$ maps to $Join(Agent, ShoppingGroup)$,
 $Join(Agent, AcademicGroup)$

3. Propositional Refinement

The process of changing the number of argument by adding more arguments,
 e.g., $Money(X, Amount)$ maps to $Money(X, Amount, Currency)$

4. Precondition Refinement

The process of changing the preconditions of an action rule, this is done by
 adding one or more preconditions to a rule,

e.g., $Buy(Agent, Book) \rightarrow Has(Agent, Book)$ maps to $Buy(Agent, Book) \wedge$
 $Instock(Book, Shop) \rightarrow Has(Agent, Book)$

Other types of mismatches

In addition to abstraction and refinement, there are some remaining types, which neither more nor less give useful information that possible to occur in ontological mismatches such as switching arguments and changing individuals. Due to our scope, we mainly focus on eight different mismatches which are predicate abstraction, domain abstraction, propositional abstraction, predicate refinement, domain refinement, propositional refinement, precondition refinement and switched argument.

3.3.2 Diagnostic Algorithm

The diagnostic algorithm is the heart of ORS. It diagnoses and identifies various types of mismatches (e.g., abstraction and refinement as we explained in Section 3.3.1). When the communication fails, ORS identifies the cause by observing the agent communication and the information from PA and plan deconstructor. In this section, we give an overview of how the diagnostic algorithm works.

Diagnostic Assumptions

There are some assumptions which make the problem tractable and enable our focus to diagnosis and repair.

- The system is designed to deal with errors on a case by case. For instance, it identifies one error, repairs it, replan and restart the cycle again to accomplish the goal again. If the failure occurs, it then diagnoses the second error.

- It assumes that the information of SPAs only receives by the question that PA asks, and then they answer back to the PA. We do not have the right to access the full SPA's ontology as we explained in the multi-agent environment (Chapter 2).
- ORS assumes that the SPAs are helpful and honest; they will attempt to perform tasks if they can do. Therefore, ORS attempts to scope the problem only either the SPAs are not able to perform actions or the SPAs' ontologies mismatch with the PA's ontology.

Fault Diagnosis

Since ORS only can access some part of information related to the problem, i.e., the question and the answer between the PA and SPAs, the key concept of fault diagnosis is *surprising questions*. The PA expects to be asked about a set of preconditions for its action to be performed. If the SPA represents these preconditions in a different way, then the query from the SPA will be differ and *surprising* to the PA. For example, if PA expects to be asked about `chooseThing(Agent,book)`, but the SPA asks about `chooseItem(Agent,book)` this is a surprising question for the PA. ORS use the surprising questions to separate the diagnosis into possible three cases as follow:

1. Failure immediately occurs after the PA makes a request to the SPA, then no query was made. The problem may be because the SPA is unable to perform the task.
2. Failure after a surprising question was asked, these preconditions and facts do not match with the PA's preconditions. The failure can be broken down into specific two cases;
 - A surprising question with an expected predicate, the following possible cases:
 - The number of arguments is different. ORS can diagnose and propose the repair as either propositional abstraction or refinement.

- The predicate's arguments are the wrong type. Domain abstraction or anti-abstract or switched arguments is used to repair this problem.
 - A surprising question with an unexpected predicate, the following possible cases:
 - There is a relation between the predicate, e.g., one is a subtype of the other. Predicate abstraction or anti-abstract can be use as a proposed repair.
 - There is no relation between the predicate which might be because of a missing precondition which can be added using precondition refinement.
3. Failure after non-surprising question was asked. There are two possible problems;
- The SPA asks a question that contains an uninstantiated variable, the PA then instantiates the variable to the wrong value.
 - The SPA asks a fully instantiated question to the PA and the PA gave the wrong truth value.

3.3.3 Repair

This is used after the diagnosis system identifies a fault in the PA's ontology. When this fault can be repaired, it also identifies what type of repair need to be perform, e.g., the abstraction or refinement. Then repair system will repair and update the PA's ontology in order to enable communication with SPAs.

However, in the current ORS, it identifies which part of ontology to be repaired and then repairs it without keeping track on what have been repaired. This might cause loss of useful information for interacting with other or previous agents that communicated successfully before changing the ontology. Therefore, the keeping the repair information in ORS will be examined in this project.

Chapter 4

Design and Implementation

As we mentioned earlier, ORS diagnose and repair within a single ontology. ORS does not keep any repair information. Consider in the multiagent system, the PA needs to communicate between several SPAs to achieve its goal. If the repairs are made to a single ontology, the agent will risk throwing away useful information. For example, if the agent interacts with SPAs that used to be successful, then after several changes the agent may be unable to communicate with them. We believe that keeping information about previous matches, which we call ‘bridge rules’ leads to less repair and decreases execution failures in the multiagent environment. Therefore, we extended ORS with utilising bridging rules to solve these problems.

In this chapter, we provide the system overview of ORS with the bridging rules component, a completed design of each sub-process and implementation.

4.1 System Overview

In addition to the ORS’s four components, this project introduces utilising bridging rules as an extended component. Therefore, there are some changes to the original flow of ORS as shown in Figure 4.1. There are two additional steps compared to the original ORS. In the original ORS, after the planning agent receives a surprising question or any *unknown term* from the service planning agent and if the PA answers no, so the action fails. This is the case when either the PA does not understand the question or it cannot satisfy the answer. The PA then calls to ORS. ORS performs the diagnosis and repairs the ontology. With the bridging rules component, when the planning agent answer no, it will always look up bridge rules that contain the

previous ontology mapping and repair information that have been made by ORS for finding any relevant possible answer to the question asked by the SPA. The bridge rules are useful in the case that the PA does not understand the question which we call an *unknown term* and this *unknown term* is linked by the bridge rule to the term that the PA can understand. Since the original ORS, we perform the diagnosis and repairs to any case that causes action failures, it might be costly. If we can find the information that can be used in the bridging rules, it would be faster and less expensive than performing the whole process of diagnosis the failures.

If the *unknown term* is found in the bridging rules, the unknown term will become a *known term* then the PA can use this known term to response the SPA and continues the action without calling ORS. If the unknown term is not found, the action will fail and then the PA calls ORS and goes through as normal ORS tasks. After the repair succeeds, the new task is to store this repair information in a bridge rule format which we can use for searching an unknown term. The details of bridging rules will be given in the next section.

With these changes to ORS, two subcomponents were created to carry out these tasks as highlighted in Figure 4.1. The first subcomponent is the *storing bridging rule* which is integrated into ORS. It is responsible for extracting the repair information from the diagnosis component and storing it as a bridge rule. The second subcomponent is the *searching bridging rule* which is called by the planning agent. This subcomponent searches bridge rules to find whether an unknown term matches in the rules or not and then returns the matched term if it is found.

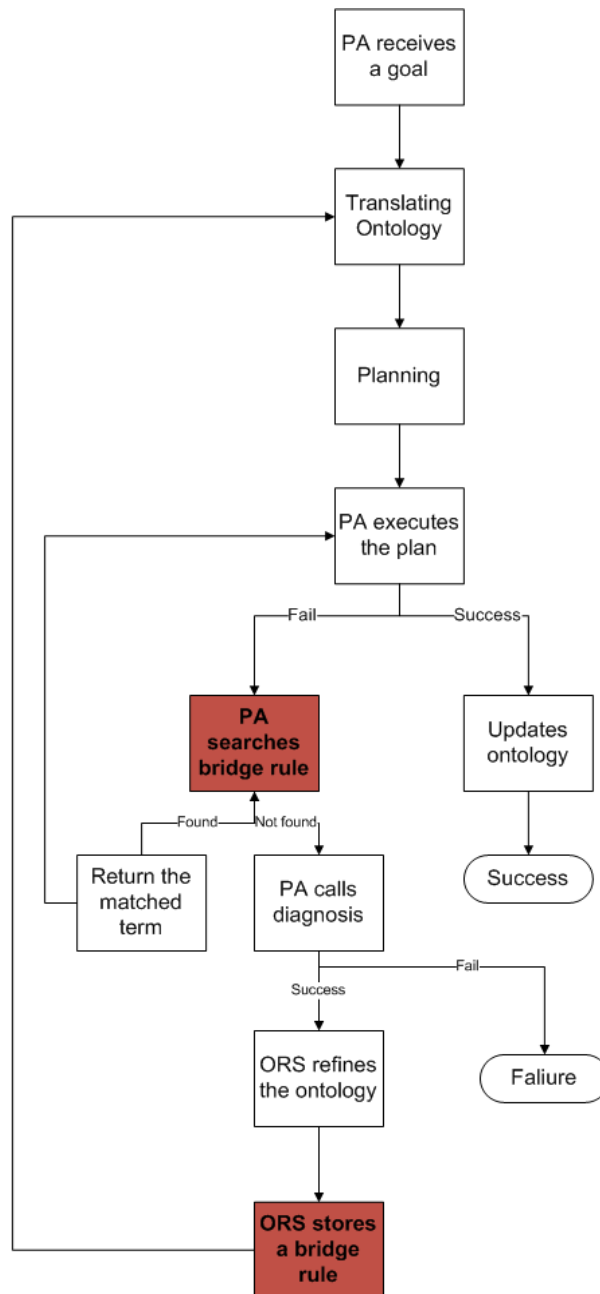


Figure 4.1: The ontology repair system flow chart with utilising bridge rules.

4.2 Bridge Rules

In order to understand the details in this project, the term “*bridge rule*” needs to be clarified. In section 2.5, we discussed several works involving bridge rules. However, those are different from what we attempt to define in this project. In this section, we describe on our view of bridge rules and its general structure.

For this project, we define bridge rules as the mapping between the original term and the updated term that is diagnosed and repaired by ORS. Therefore, this mapping is similar to a bridge that connects between two different terms. The mapping information can be received by the diagnosis component and extracted into a bridge rule structure. The details will be given in section 4.3.

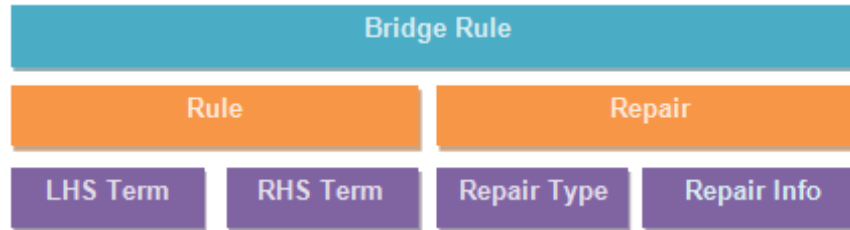


Figure 4.2: The structure of a bridge rule.

To follow Figure 4.2, the following is the example of a bridge rule:

[money(aibook,dollar,100),money(aibook,100)],[propositionalAA,[money,2,[book,number],currency,2]].

A bridge rule consists of two main sections: *rule* and *repair* information as shown in figure 4.2. The rule section is the main part for the mapping process, it contains two terms: the original term and the updated term. In the example, if *money(aibook,100)* was in the PA's ontology and after ORS diagnoses and repairs it with *money(aibook, dollar,100)* then the rule section will be:

money(aibook,100) → money(aibook,dollar,100).

This rule means that *money(aibook,100)* is replaced by *money(aibook,dollar,100)* after repairing ontology. Therefore, *money(aibook,dollar,100)* is the updated term in a current version of the PA's ontology. The former term *money(aibook,100)* is called as the *original term* or the *left hand-side (LHS) term*. The latter term *money(aibook,dollar,100)* is called as the *updated term* or the *right hand-side (RHS) term*. Therefore, if the planning agent receives the query about *money(aibook,100)*, it then can look up this bridge rule and return *money(aibook,100)* back to the service providing agent even if it does not exist in its ontology but when the RHS term is found in the its ontology. We will discuss this task in section 4.4.

To perform the mapping and return $money(aibook,100)$, the *repair information* is necessary. The repair information is the information that is received from the diagnosis component. In the example, the diagnosis component identified that it is the *propositional refinement* by adding more argument to the term. The repair information is composed of the information of the original term before it is repaired together with the changes depending on each mismatch type. The general repair information contains in the following order: repair type, predicate name, arity, predicate's class arguments, and additional repair information for each repair type. Therefore, the repair information for this example will be as follow:

- Repair Type: **precondAA**
- Predicate name : **money**
- Arity : **2**
- Predicate's class arguments : **book, number**
- Addition repair information: **currency,2** (added argument name, its position)

This example is only specific for the propositional refinement type; additional repair information field may take different values and can be divided into different subsections depending on the repair type. The details of different repair information will be explained in section 4.3.

4.3 Storing Bridge Rules

As discussed earlier, we believe that repair information might be useful for future communication in the multiagent system. We decided to store this information in a structure that we call the bridge rule. To store a bridge rule, we need to identify what information that we can receive from the repair component and match them with our bridge rule concept that was introduced in the previous section. In this section, we provide the description on how we design and implement the storing for bridge rules.

During the diagnosis and repair processes, ORS keeps track of the *repair type* and *repair information* and reports these values back to the planning agent. The storing bridge rules process will be called by ORS after repair is perform successfully. It requests the repair information and extracts to a bridge rule structure depending on its repair type. There are eight repair types which we consider storing in the bridge

rule. Before we provide our bridge rule formats, the repair information from ORS needs to be extracted and the values they contain needs to be identified.

4.3.1 ORS Repair Information

The repair information is composed of 3 mains subsections; *the original term*, *updated term* and *relevant repair information*. The followings are the repair information structures from diagnosis component together with their examples:

1. Switch Argument

Repair = [original term, updated term, [**switchArgs**, predicate name, switch list, argument classes list]].

Example:

```
[interest(shoppingAgent,bookShopGroup),interest(bookShopGroup,shoppingAgent),[switchArgs,[shoppingGroup,agent],interest,[[shoppingGroup,agent],[agent,shoppingGroup]]]]
```

The main subsection in switch argument type is a *switch list*. Switch list is a list contains the predicate's argument classes of the switched arguments to identify which the argument pair is switched before and after. In the example, the switch list is `[[shoppingGroup,agent],[agent,shoppingGroup]]` that tell *shoppingGroup* and *agent* are the switched arguments.

2. Domain Abstraction

Repair = [original term, updated term, [**domainA**, [argument classes list, my unmatched class, his unmatched class]]]

Example:

```
[chooseThing(shoppingAgent,titchmarshMemoirs),chooseThing(shoppingAgent,ourMutualFriend),[domainA,[[gardeningBook,agent],book,gardeningBook]]]
```

The domain abstraction changes the domain of the argument. The repair information keeps the unmatched class of the planning agent and the service providing agent. In the example, *chooseThing(agent,book)* becomes *chooseThing(agent,gardeningBook)*. The additional information is the PA's

unmatched class and the SPA's unmatched class which could tell us about the subtype relationship. For the domain abstraction, the SPA's unmatched class (*gardeningBook*) is a subtype of the PA's unmatched class (*book*).

3. Domain Refinement

Repair = [original term, updated term, [**domainAA**, [argument classes list, my argument class, his argument class]]]

Example:

[chooseThing(shoppingAgent,ourMutualFriend),chooseThing(shoppingAgent,titchmarshMemoirs),[domainAA,[[book,agent],gardeningBook,book]]]

This type is similar to the domain abstraction. The only thing is how we represent the subtype relation. For the domain refinement, the PA's unmatched class (*gardeningBook*) is a subtype of the SPA's unmatched class (*book*).

4. Predicate Abstraction

Repair = [original term, updated term,[**predicateA**,[updated predicate name, original predicate name]]]

Example:

[chooseItem(shoppingAgent,ourMutualFriend),chooseThing(shoppingAgent,ourMutualFriend),[predicateA,[chooseThing,chooseItem]]]

The main repair information is that the predicate name is changed in more general term. It keeps two predicate names; before changing one and after changing one as the additional information for this type. In the example, *chooseThing* is the changed predicate and *chooseItem* is the original one.

5. Predicate Refinement

Repair = [original term, updated term, [**predicateAA**, [updated predicate name, original predicate name]]]

Example:

[chooseThing(shoppingAgent,ourMutualFriend),chooseItem(shoppingAgent,ourMutualFriend), [predicateAA, [chooseItem,chooseThing]]]

This type keeps the same structure as a predicate abstraction because it changes only the predicate name. Although they look the same in practice but they contain different definitions in theory.

6. Propositional Abstraction

Repair = [original term, updated term, [**propositionalA**, [argument class, Position]]].

Example:

[chooseItem(shoppingAgent,ourMutualFriend,10),chooseItem(shoppingAgent,ourMutualFriend),[propositionalA,[number,3]]]

The additional repair information of propositional abstraction is the argument class and position that is dropped from the predicate. In this example, the argument class is a number which locates in the third argument.

7. Propositional Refinement

Repair = [original term, updated term, [**propositionalAA**, argument class, Position]]].

Example:

[money(shoppingAgent,100),money(shoppingAgent,dollar,100),[propositionalAA,[currency,2]]]

The specific information for propositional abstraction is the argument class and position that is added from the predicate which is similar to the propositional abstraction.

8. Class Precondition Refinement

Repair = [WrongClass,RightClass,[**precondAA**,class]].

Example:

The PA's ontology rule :

$\text{joinGroup}(\text{Agent},\text{Group}) \rightarrow \text{class}(\text{shoppingAgent},\text{Agent}) \wedge \text{class}(\text{aiGroup},\text{Group})$

The SPA asks for $\text{class}(\text{aiGroup},\text{shopping})$ then the repair information is

[group,shoppingGroup, [precondAA,class]]

Our scope focuses on the class precondition. This repair type contains different information compared to other types. In the action rule, it contains several preconditions and class precondition that state each argument in the rule should belong to which class. However, there is a case when the service providing agent asks for different class condition. If ORS can diagnose and it will repair the ontology by adding the class condition to precondition of the action rule. In the example, `class(aiGroup,shoppingGroup)` is adding to precondition of `joinGroup(Agent,Group)`. Then the repair information contains wrong class and right class for the argument.

ORS and its agent environments are written in Prolog. Although the ontology is written using KIF representation, during the execution, the ontology is translated into the Prolog ontology. This is because the PA and ORS can most easily work and access with the Prolog ontology. In this implementation, it is reasonable to keep a bridge rule using Prolog representation so it can be easily accessed by the ORS. Moreover, the logic programming language such as Prolog allows us to query the answer from the existing predicate in the file. Each predicate corresponds to a record or a row and its argument values correspond to values in columns. Therefore, a collection of bridge rules is stored in a single file which we called the “bridge rule database”, this database is a Prolog based file. This database contains a collection of *bridgeRule* predicate.

4.3.2 Bridge Rules

Each bridge rule consists of repair information in the form:

bridgeRule([Original Term, Updated Term], [Repair Type,[Repair Information]]).

The values inside a predicate are extracted from the repair information given by ORS as we discussed earlier. However, a bridge rule contains slightly different information because the purpose of keeping this information is to be able to reuse and backtrack the original term rather than report the repair result. Each repair type is kept in the database as follows:

1. Switch Argument

bridgeRule(Original Term, Updated Term, [switchArgs, [Predicate name, Arity, [Argument Class list], [Switch list]])

Example:

The rule is:

interest(bookShopGroup, shoppingAgent) → interest(shoppingAgent, bookShopGroup)

bridgeRule([interest(bookShopGroup,shoppingAgent),interest(shoppingAgent, bookShopGroup)],[switchArgs,[interest,2,[shoppingGroup,agent],[agent,shoppingGroup],[shoppingGroup,agent]]]).

To store this structure, the repair information from diagnosis component contains all values that we need.

2. Domain Abstraction

bridgeRule(Original Term, Updated Term, [domainA, [Predicate name, Arity, [Argument Class list],changed class, subclass(SPA's unmatched class, PA's unmatched class)]])

Example:

The rule is:

chooseThing(shoppingAgent, titchmarshMemoirs)→
chooseThing(shoppingAgent,ourMutualFriend)

bridgeRule([chooseThing(shoppingAgent, titchmarshMemoirs), chooseThing (shoppingAgent,ourMutualFriend)],[domainA,[chooseThing,2,[agent,gardenin gBook],book,subclass(gardeningBook,book)]]).

For domain abstraction, we store the bridge rule in a different format from repair result. The changed class is kept as the argument after the argument class list. We also transform both unmatched into subtype relation and keep value as *subclass(X, Y)*, where X is the SPA's unmatched class and Y is the PA's unmatched class that extract for repair result.

3. Domain Refinement

bridgeRule(Original Term, Updated Term, [domainAA, [Predicate name, Arity, [Argument Class list],changed class, subclass(PA's unmatched class, SPA's unmatched class)]])

Example:

The rule is:

chooseThing(shoppingAgent,ourMutualFriend)→
chooseThing(shoppingAgent,titchmarshMemoirs)

bridgeRule([chooseThing(shoppingAgent,ourMutualFriend),chooseThing(shoppingAgent,titchmarshMemoirs)],[domainAA,[chooseThing,2,[agent,book],gardeningBook,subclass(gardeningBook,book)]]).

This type keeps the same format as domain abstraction. However, the only different is subtype relation *subclass(X, Y)* which is stored in the opposite way, i.e., *X* is the PA's unmatched class and *Y* is the SPA's unmatched class that extract for repair result.

4. Predicate Abstraction

bridgeRule(Original Term, Updated Term,[predicateA,[Before Predicate name, Arity,[Argument Class list]])

Example:

The rule is:

chooseItem(shoppingAgent,ourMutualFriend)→
chooseThing(shoppingAgent,ourMutualFriend)

bridgeRule([chooseItem(shoppingAgent,ourMutualFriend),chooseThing(shoppingAgent,ourMutualFriend)],[predicateA,[chooseItem,2,[agent,book]])].

If we compare with the repair result, the bridge rule only keep the before changed predicate name instead of keeping both of them because we already keep the original and updated term.

5. Predicate Refinement

bridgeRule(Original Term, Updated Term,[predicateAA,[Before Predicate name, Arity,[Argument Class list]]])

Example:

The rule is:

chooseThing(shoppingAgent,ourMutualFriend)→
chooseItem(shoppingAgent,ourMutualFriend)

*bridgeRule([chooseThing(shoppingAgent,ourMutualFriend),chooseItem(shoppingAgent,ourMutualFriend)],
[predicateAA,[chooseThing,2,[agent,book]]]).*

Predicate refinement contains the same structure as provided in predicate abstraction.

6. Propositional Abstraction

bridgeRule(Original Term, Updated Term,[propositionalA,[Predicate name, Before changed arity, [Argument Class list], Position]])

Example:

The rule is:

money(shoppingAgent,dollars,100)→money(shoppingAgent,100)

*bridgeRule([money(shoppingAgent,dollars,100),money(shoppingAgent,100)],
[propositionalA,[money,3,[agent,currency,uninstantiated],currency,2]]).*

This structure covers all information that contain in the repair result. We focus on the information of the term before it is changes. Therefore we only keep the arity and argument class list of the original term.

7. Propositional Refinement

bridgeRule(Original Term, Updated Term,[propositionalAA,[Predicate name, Before changed arity, [Argument Class list], Position]])

Example:

The rule is:

$\text{money}(\text{shoppingAgent}, 100) \rightarrow \text{money}(\text{shoppingAgent}, \text{dollar}, 100)$

$\text{bridgeRule}([\text{money}(\text{shoppingAgent}, 100), \text{money}(\text{shoppingAgent}, \text{dollar}, 100)], [\text{propositionalAA}, [\text{money}, 2, [\text{agent}, \text{number}], \text{currency}, 2]])$.

This structure is similar to propositional abstraction as discussed before.

8. Precondition Refinement

$\text{bridgeRule}([\text{wrong class}, \text{right class}], [\text{precondAA}, [\text{class}, 2, \text{action}]])$.

Example:

The rule is:

$\text{putInBasket}(\text{shoppingAgent}, \text{aiGroup}, \text{ourMutualFriend}) \rightarrow$

$\text{class}(\text{shoppingAgent}, \text{agent}) \wedge \text{class}(\text{aiGroup}, \text{group}) \wedge$

$\text{class}(\text{ourMutualFriend}, \text{book})$ maps to

$\text{putInBasket}(\text{shoppingAgent}, \text{aiGroup}, \text{ourMutualFriend}) \rightarrow$

$\text{class}(\text{shoppingAgent}, \text{agent}) \wedge \text{class}(\text{aiGroup}, \text{group}) \wedge \text{class}(\text{aiGroup}, \text{shoppingGroup}) \wedge \text{class}(\text{ourMutualFriend}, \text{book})$

$\text{bridgeRule}([\text{group}, \text{shoppingGroup}], [\text{precondAA}, [\text{class}, 2, \text{putInBasket}(\text{shoppingAgent}, \text{aiGroup}, \text{ourMutualFriend})]])$

As we discussed before, this repair type is different from others. Therefore, the information contains in this type and different meaning. It composed of the *wrong class*, the *right class* that needs to be added in the precondition and the repair information contain *class* which tells this is a class precondition refinement, number 2 is an arity of class predicate and the action rule that this precondition is added. However, we decided not to store this type in a bridge rule because we found that there is not much advantage to storing it. Details will discuss in the section 4.4.

These eight structures of a bridge rule will be selected depending on which type of repair is diagnosed and repaired by ORS. After the repair component performs successfully, ORS sends the repair result to store bridge rule component. The store component first extracts a repair type. Then it passes the repair result to construct a bridge rule format based on its type as we provided earlier. The next step, it then

records this bridge rule into a bridge rule database which will be accessed by the planning agent.

4.4 Searching and Consuming Bridge Rules

According to our hypothesis, keeping bridge rules leads to less repair and decreases execution failure, we discussed how to keep bridge rules in section 4.3. Therefore, in this section will describe how we can access and retrieve a bridge rule during execution of the system to decrease action failures.

The searching process is called by the planning agent when it is trying to answer the surprising question that is asked by the service providing agent. To scope our project, we only consider the surprising question about the ontological mismatch. For example, the SPA asks the PA about $money(shoppingAgent,100)$, but the PA answer no. This case may be either the PA does not understand $money(shoppingAgent,100)$ or it cannot satisfy this term to SPA. With bridging rules, we assume that $money(shoppingAgent,100)$ is an *unknown term*. Before the PA consults ORS, the PA will always looks up the bridge rule to attempt to match this unknown term with any existing rule in the database. If it is found, the planning agent can use this term and continue execution the plan. To search and match the unknown term, we need access to the current PA's ontology, to check it against the information whether it contains the RHS term as stored in a bridge rule. Otherwise, it returns "*not found*".

As we stored a bridge rule depending on its repair type, the method for searching this unknown term is also implemented based on its repair type. However, the general steps on how to verify the unknown as shown in Figure 4.3. To find the matched term, it has to pass several checks. Our goal is to match the unknown term to the LHS term of a bridge rule. First, we retrieve the unknown term, which consists of predicate name and instantiated argument. The predicate name is checked to see whether it matches a rule in the bridge rule database. Then it compares the arity and argument class list with the values in the bridge rule. If these comparisons show that the arity and arguments equal, we then check the PA's ontology based on the repair information in each type and the RHS term that they should exist in the PA's ontology. If all of them are matched, it means the planning agent knows the *unknown*

term and this unknown term will become the *known term* and answer back to the service providing agent. Otherwise, it returns term is not found and continues to diagnosis component.

We need different methods to perform on checking the planning agent's ontology, for example, the domain abstraction needs to check whether the ontology contain the same subclass relation as in the repair information in addition to the same RHS term. Thus, the different matches are classified as follows:

1. Switch argument

This type uses the *switch list* as extra checking information. It checks whether the ontology contains this argument list in the switch list or not.

2. Predicate Abstraction/ Refinement

The types only change the name of the predicate. Therefore, the argument classes need to be the same as before and after changes. Then we check the arguments of unknown term with the current argument of the term in the ontology.

3. Propositional Abstraction/ Refinement

The types add or drop one of the arguments in the predicate. The bridge rule keeps the name of this argument. We need to ensure that the ontology contains this argument which means this argument class has to be found in the PA's ontology.

4. Domain Abstraction/Refinement

The extra information is a subclass relationship. The ontology has to contain the same subclass relationship as in a bridge rule.

5. Precondition Refinement

The precondition contains different type of mismatch. As mentioned earlier, we possible store this information but there is not much advantage to having the bridge rule. We consider only the class precondition that has the form of $class(X, Y)$ means X is in class Y together with its action rule. We could check in bridge rules that it contains any information about class Y. If Y is found in the LHS term, then we check RHS class whether it contains in the ontology with the same action rule and also check that the PA knows $class(X, Y)$ then it is able to use this class as one of the preconditions. However, this

case will never be true because if the PA have known $class(X, Y)$, it would never have answered “no” to a query. Therefore, most cases of the bridge rules should fail because we cannot find the $class(X, Y)$ in the current PA’s ontology.

These checks are to ensure that the planning agent is used to know and contain this unknown term before so that it can be sent back as the known term to answer the service providing agent and continue the action without failure. However, there is possible that the ontological mismatch is combined from two or more types of mismatch at the same time. This type is a *compound mismatch* which the current ORS cannot identify and repair this type. Therefore, the bridge rules can identify the compound mismatch if they contain all the related repair types in the database. We first follow the step for search in bridge rules. If we cannot find any match, we could try to find the rules that might link together and create the new rule. We will check this rule and compare with the PA’s ontology following the same process in Figure 4.3 and return the result. For example, the bridge rules are:

$$money(Agent, Amount) \rightarrow money(Agent, Currency, Amount)$$

$$money(Agent, Currency, Amount) \rightarrow cash(Agent, Currency, Amount)$$

If the PA receives a query about $money(Agent, Amount)$ and it answers no, it will look up bridge rules to find the answer. If it could match this term with the LHS term in bridge rules, but the RHS term is not in the ontology, there could be two cases: it cannot find in the ontology or it is a compound mismatch. Therefore, it searches again for any link between bridge rules and it could find two bridge rules and combine them into the new rule:

$$money(Agent, Amount) \rightarrow cash(Agent, Currency, Amount)$$

Then it checks the RHS term, $cash(Agent, Currency, Amount)$, to see whether it is in the ontology or not and returns the result as shown in Figure 4.3. We will discuss our result of this implementation in the next chapter.

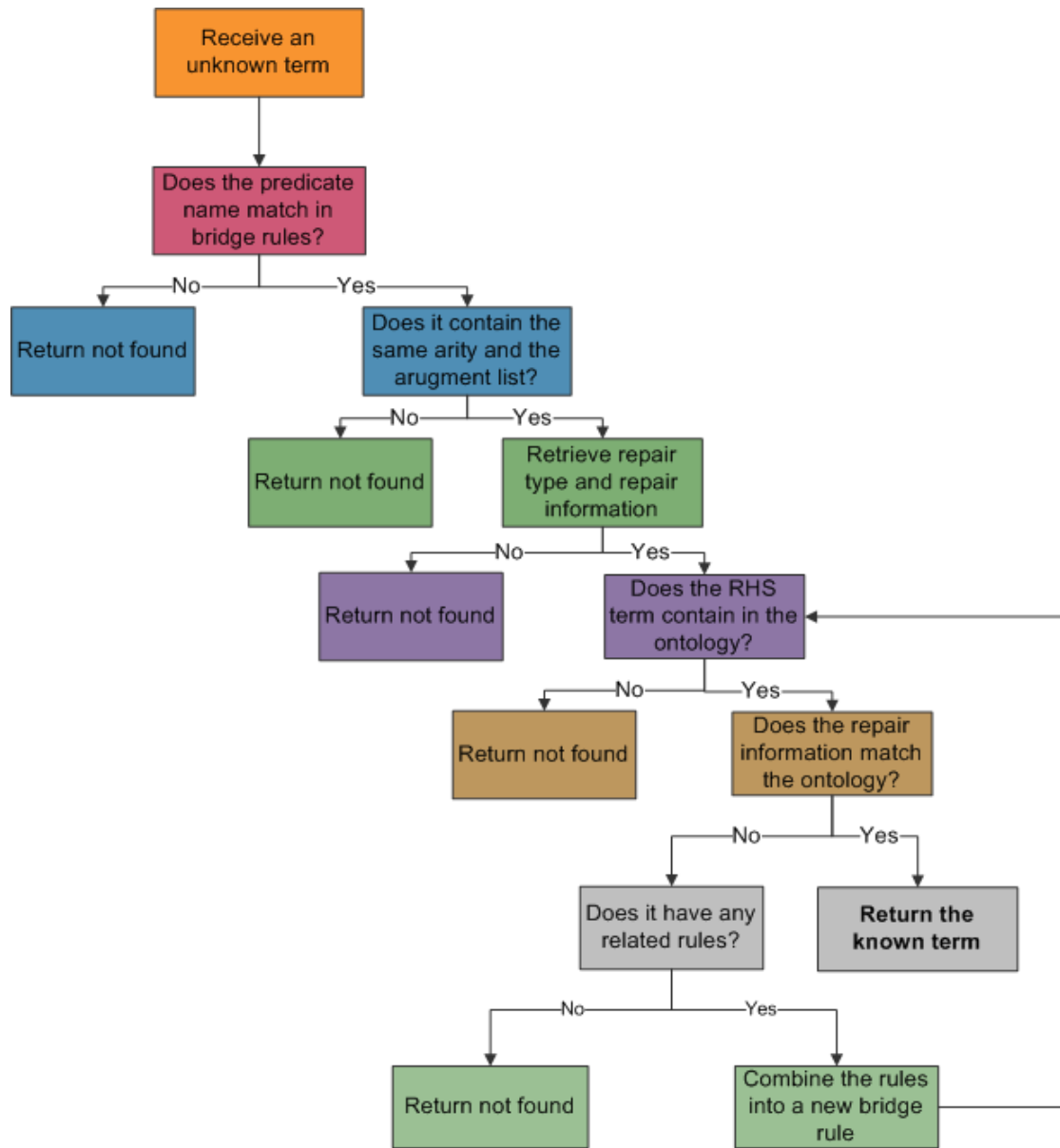


Figure 4.3: The overview of searching bridge rule method.

Chapter 5

Evaluation

The evaluation helps us to conclude whether our bridge rules are worth storing, proving the result described our hypothesis. We recall our hypothesis is:

“Keeping the bridge rules of previous matches performed by ORS leads to less repair and decreases execution failures in multiagent systems compared to ORS without the bridge rules.”

Therefore, we could test our version against the original ORS with the existing scenario to see the differences between the versions. In this chapter, we explain how we set up the evaluation including explaining the goals and the scenario, and provide and analyse the results.

5.1 Effectiveness and Efficiency

Before we start the evaluation, we need to specify what the goals that we want to test are and what kind of results we expect to see. The general goals are to see the system improvement in both effectiveness and efficiency. By *effectiveness*, we mean how well the system work which should correspond to the hypothesis. Whereas, efficiency refers to the running time compared to the original ORS.

For evaluating the effectiveness, we provided examples of the ontological mismatches together with their corresponding bridge rules in the database. If our new component can search and find this mismatch term and return to the PA and continue using this term without failure, we can conclude that our new component is effective compared with the original system. Because our hypothesis is that when we have

bridge rules and these bridge rules can be matched with the term that the PA does not understand, this term can be used and the diagnosis component will not be called and the action can continue without failure. Whereas in the original system, when the PA answer “no” which could be either it cannot satisfy or it does not understand a query, its action then fails, the PA asks ORS to perform a diagnosis and the PA need to start the whole cycle again with planning the actions and carrying out them. Another challenge that we could examine is the effectiveness is the compound mismatch. When two or more mismatches occur in a single case, the original ORS cannot repair it. We assume that ORS alters the PA’s ontology several times and these repair information are stored in bridge rules, then we can possible find the link between bridge rules and create the new rule by combining them together. In the evaluation, we believe that having two or more bridge rules in that are related to the compound mismatch, we expect them to be linked and return the possible term in bridge rules. Thus, we will set up a test case to examine the problem of compound mismatch later on in the chapter.

Another goal is efficiency. We focus on the cost of integrating this component by measuring the time it took the system to run the whole processes and compare it with the original system. Since we introduced new tasks to the ORS, i.e., searching the bridge rules and storing bridge rules we need to see how much time that we need to searching and storing them compared with the system without bridge rules. There are several cases that we can examine. First, we can evaluate the time with using bridge rule against performing the diagnosis and repair components. If the term is found in the bridge rule, it should be faster than running the diagnosis. Second, we can evaluate how much it does waste to look up and store bridging rules every time the PA answers “no” and it cannot find any match to bridge rules. In this case, we might expect with less efficiency to the original system.

Moreover, measuring the time is a very sensitive task. There are several factors that can make a variation in the results such as using a different machine, running a different number of programs during the testing process, and lagging time during agent communication. Therefore, we address these problems by using the same machine for all evaluations and perform ten different runs and averaging the results

to eliminate the variance. The details of test cases and evaluation processes will be discussed later in Section 5.3.

5.2 The Shopping Ontology

To evaluate the extended version of ORS, we develop test cases based on the shopping ontology. The shopping ontology is a sample of an ontology for an online bookstore which is one of the existing ontologies in ORS. This ontology was used in the evaluation of the original system. Since this ontology is in the general domain that people already know about shopping, it may be easier to understand and demonstrate the problem of ontological mismatch than one from a specific domain such as Biology. Therefore, we decided to use this ontology to evaluate the new version so it would be easier to compare with the original system.

The shopping ontology contains five different SPAs and the PA together with the environment for interacting between SPAs and the PA. The SPAs are two *join group* agents, two *put in basket* agents, and a *buy* agent. Each of the SPAs has its own ontology that contains preconditions of the actions for their specific tasks. For example, the *put in basket* agent performs the action of putting the book into the basket. In this scenario, the goal of the PA is to buy the book “Our Mutual Friend”. The goal includes actions from a different type of SPA such as choosing the correct book, obtaining membership of the relevant group, putting the book into the shopping basket, buying the book, and etc.

Although we develop test cases that may contain different actions and different term compared to the original shopping ontology, but all test cases are created to achieve the same goal which is to purchase the book “Our Mutual Friend”. We will explain our modified shopping ontology in the next section.

5.3 The Experiment

We developed three ontologies based on the shopping ontology which they contain ontological mismatches. We also created three corresponding bridge rule databases which contain different set of bridge rules to demonstrate possible scenarios of using

bridge rules to examine the effectiveness of the extended ORS. In this section, we explain our ontologies, bridge rules database and test cases.

5.3.1 Ontologies

As mentioned earlier, we have created three ontologies: Shopping A, Shopping B, and Shopping C. These ontologies have the same goal as the original shopping ontology, i.e., to purchase the book “Our Mutual Friend”. Each ontology contains several ontological mismatches, as shown in Figure 5.1. The following are the mismatches types that we consider in this evaluation:

1. Class Precondition Refinement

$$putInBasket(Agent, Book, Group) \rightarrow putInBasket(Agent, Book, Group) \wedge class(Group, shoppingGroup)$$

The class precondition refinement is a mismatch between the *put in basket* agent and the PA. The *put in basket* request an argument of the *putInBasket* action rule to be in the *shoppingGroup* class but in the PA’s ontology represents this argument in the *group* class. Therefore, we need to add the class precondition *class(Group, shoppingGroup)* into this action.

2. Propositional Refinement

$$money(Agent, Amount) \rightarrow money(Agent, Currency, Amount)$$

The mismatch occurs with the *buy* agent. It requests *Money(Agent, Currency, Amount)* as the precondition of buy action. Instead, the PA contains *Money(Agent, Amount)* in its ontology.

3. Predicate Refinement

$$chooseItem(Agent, Book) \rightarrow chooseThing(Agent, Book)$$

The mismatch also occurs with the *put in basket* agent as it requires one of the preconditions to be *chooseThing(Agent, Book)*. In contrast, the PA contains *chooseItem(Agent, Book)* in its ontology.

4. Switch Argument

$$interest(Agent, Group) \rightarrow interest(Group, Agent)$$

The mismatch occurs with the *join group* agent while it *requests interest(Group, Agent)* as one of the preconditions instead of *interest(Agent, Group)* as in the PA's ontology.

5. Compound Mismatch

$chooseItem(Agent, Book, Ref) \rightarrow chooseThing(Agent, Book)$

=

$chooseItem(Agent, Book, Ref) \rightarrow ChooseItem(Agent, Book)$

+

$chooseItem(Agent, Book) \rightarrow chooseThing(Agent, Book)$

This compound mismatch is in Shopping C. The *put in basket* requests *chooseThing(Agent,Book)* but in the PA's ontology contains *chooseItem(Agent,Book,Ref)*. This mismatch is the combination of the predicate refinement and propositional abstraction. Due to the limit of the original ORS, it cannot be diagnosed. Therefore, we want to test if it is possible to use bridge rule to address this problem.

Shopping A, only consists of ontological mismatches 2 and 3. However, Shopping B contains ontological mismatches 1, 2, 3 and 4. Finally, Shopping C consists of ontological mismatches 2, 3 and 5.

Shopping A	<ul style="list-style-type: none"> • Predication Refinement • Propositional Refinement
Shopping B	<ul style="list-style-type: none"> • Class Precondition Refinement • Predication Refinement • Propositional Refinement • Switch Argument
Shopping C	<ul style="list-style-type: none"> • Predication Refinement • Propositional Refinement • Compound Mismatch

Figure 5.1 : The different mismatches between three shopping ontologies.

5.3.2 Bridge Rules

We created three different sets of bridge rules called Set A, Set B and Set C. Each bridge rule corresponds with our shopping ontologies. We manually create these sets and assume that these bridge rules contain repair information that were previously successful repaired by ORS. The corresponding bridge rules contain repair information and the repair types with the shopping ontologies.

For example, we suppose that ORS repaired the propositional abstraction of :

$$money(Agent,Currency,Amount) \rightarrow money(Agent,Amount)$$

This information is stored in the bridge rule as discussed in Chapter 4. Then we assume that we start the communication again and this time the SPA requests $money(Agent,Currency,Amount)$ which the PA answers “no” and look up the bridge rule. It searches the bridge rules to find any rule that contains the LHS term as $money(Agent,Currency,Amount)$. In this example, the rule above is found. We then check the current PA’s ontology to see whether it contains the RHS term, $money(Agent,Amount)$, or not. Details of searching bridge rules were discussed in Chapter 4. In contrast, if the bridge rules return “not found”, the action fails and the PA call ORS for diagnosis and repairing it as propositional refinement. Therefore, to perform matching in bridge rules, the information need to be opposite repair type with the ontology.

The following are the sets of bridge rules that we created for testing our system, details can be found in Appendix B:

- Set A, contains a set of bridge rules that should be fully matched with Shopping A. This is because we want to evaluate the effectiveness of the extended ORS. Since this set is exactly created to match with Shopping A, we expect the result: no failure and no repair in the system.
- Set B, it comprises of bridge rules that should be partially matched with Shopping B. We want to see how well ORS works together with the new component. The expected result would be if the bridge rule can be used then the action will continues without failure until it cannot be used. Otherwise,

the plan fails and the PA calls ORS to diagnose and repair the ontology then the repair information needs to be extracted and stored into a bridge rule. This case could be matched with our hypothesis that keeping bridge rule leads to less failure and less repair.

- Set C, contains a set of bridge rules that should be fully matched with a single type of mismatches in Shopping C. However, the challenge is that we want to evaluate is the compound mismatch, we then stored two bridge rules which are predicate refinement and propositional refinement that could be used for matching the compound mismatch in Shopping C. We expect that using bridge rules could find and combine the rule and return the result.

5.3.3 Test cases for evaluations

For examining effectiveness, we constructed nine different test cases by crossing product of our three shopping ontologies and three sets of bridge rules. The result that we expect to see is that the number of repair and the number of plan failures decrease compared with the ORS without bridge rules.

For evaluating efficiency, we consider two cases as mentioned in section 5.1. Firstly, we examine the running time of using bridge rules compared with the original ORS. Our hypothesis is that using bridge rules is faster than performing diagnosis and repairing the ontology. Then we use the running time of nine test cases compared with the original ORS. To eliminate the variance, each test case has been run ten times and the results are averaged from these trials. Secondly, we examine how much time we waste by always searching and storing bridge rule compared with the original ORS. In this case we evaluate the extended ORS by using three shopping ontologies without a set of bridge rules and compared with the running time with the original ORS. Therefore, we have the total twelve test cases as shown in Figure 5.2 and Figure 5.3.

Case 1	Shopping A Set A	Case 4	Shopping B Set A	Case 7	Shopping C Set A
Case 2	Shopping A Set B	Case 5	Shopping B Set B	Case 8	Shopping C Set B
Case 3	Shopping A Set C	Case 6	Shopping B Set C	Case 9	Shopping C Set C

Figure 5.2: The nine test cases from a combination of each ontology and bridge rule.

Case 10	Shopping A	Case 11	Shopping B	Case 12	Shopping C
----------------	------------	----------------	------------	----------------	------------

Figure 5.3: The three test cases without sets of bridge rules.

The scenarios in case 1, 5 and 9 are the cases that match with our explanations in Section 5.3.2 because it consists of the shopping ontology and its corresponding set of bridge rules.

5.4 Results

After testing the system, we provide the results in this section before analysing them in the next section. We label the original ORS as *ORS* and ORS with bridge rules as *Case X* for using later in this section. Each test case was run ten times to eliminate the variance of the results as shown in table 5.1. To better visualise and analyse the data, we also use the line graph to represent the results.

Trial	Shopping A				
	ORS	Case 1	Case 2	Case 3	Case 10
1	24.2	9.36	14.92	9.49	24.54
2	24.49	9.36	14.91	9.42	24.37
3	24.29	9.36	14.91	9.36	24.42
4	24.27	9.43	14.81	9.37	24.29
5	24.44	9.44	14.85	9.4	24.36
6	24.22	9.38	15.07	9.35	24.37
7	24.2	9.4	14.94	9.41	24.66
8	24.2	9.34	14.84	9.36	24.42
9	24.27	9.4	14.87	9.37	24.58
10	23.56	9.4	14.98	9.41	24.39
Average	24.214	9.387	14.91	9.394	24.44

Trial	Shopping B				
	ORS	Case4	Case 5	Case 6	Case 11
1	29.8	19.21	21.64	19.34	30.18
2	29.9	19.32	21.49	19.31	30.11
3	30.04	19.57	21.46	19.51	30.23
4	29.85	19.33	21.45	19.45	30.36
5	29.83	19.42	21.44	19.42	30.21
6	29.9	19.35	21.51	19.32	30.09
7	29.84	19.42	21.58	19.42	30.06
8	29.86	19.54	21.44	19.38	30.11
9	29.83	19.39	21.45	19.4	30.11
10	29.84	19.38	21.52	19.69	30.12
Average	29.869	19.39	21.498	19.42	30.158

Trial	Shopping C				
	ORS	Case 7	Case 8	Case 9	Case 12
1	Failed	Failed	Failed	9.47	Failed
2	Failed	Failed	Failed	9.42	Failed
3	Failed	Failed	Failed	9.52	Failed
4	Failed	Failed	Failed	9.48	Failed
5	Failed	Failed	Failed	9.47	Failed
6	Failed	Failed	Failed	9.47	Failed
7	Failed	Failed	Failed	8.61	Failed
8	Failed	Failed	Failed	9.44	Failed
9	Failed	Failed	Failed	9.49	Failed
10	Failed	Failed	Failed	9.47	Failed
Average	-	-	-	9.384	-

Table 5.1: The tables of running times in second for each test case.

Shopping A					Shopping B					Shopping C				
ORS	Case 1	Case 2	Case 3	Case 10	ORS	Case4	Case 5	Case 6	Case 11	ORS	Case 7	Case 8	Case 9	Case 12
2	0	1	0	2	4	2	2	2	4	Failed	Failed	Failed	0	Failed

Table 5.2: The table of the number of the internal action failed and performing repair before achieving the goal. For Shopping C, the system could not achieve the goal, except Case 9.

5.4.1 Raw results

We performed ten trials for each test set to capture the execution times. The results are shown in Table 5.1 which the average results. We can observe that all test cases that contain a set of bridge rules are faster than the original ORS. It tends to be faster

if it could find all the terms that the PA does not understand in bridge rules. In Table 5.2, we capture how many times the plan failed during the execution and performed diagnosis and then repaired ontology. The original ORS had the highest number than this other test cases with bridge rules. As our hypothesis, that shows that the bridge rule component could lead to less failure and repair. It also captured the problem of compound mismatch that is a currently limitation in ORS. Thus, we will analyse this results in term of effectiveness and efficiency later in section 5.5.

5.4.2 Line graph

From the raw results, we use the line graph to visualise them. Since it may be difficult to analyse and see the different by using the raw data, the line graph might give us better views that we can easier see how much it changes by using bridge rule. Therefore, we created two graphs for each set of results which are the normal line graph from the raw results and the cumulative line graph to see more views.

The results from Shopping A are shown in a line graph and cumulative line graph as in Figure 5.4 and Figure 5.5. For Shopping B, the results are as in Figure 5.6 and Figure 5.7. We can see that the original ORS is more than twice as slow as the bridge rules. However, we cannot compare the execution time of Shopping C because all the test cases failed except the case 9. Moreover, there is another scheme that we want to evaluate which is how much time it cost to search and store bridging rules if there is no information that we can find compared to the original ORS. The results are shown in Figure 5.8 and Figure 5.9 for Shopping A and in Figure 5.10 and Figure 5.11 for Shopping B which we can observe from the graphs that searching and storing bridge rules does not waste much time as they are almost the same lines. The details of evaluation and analysis these graphs will be given in the section 5.5.

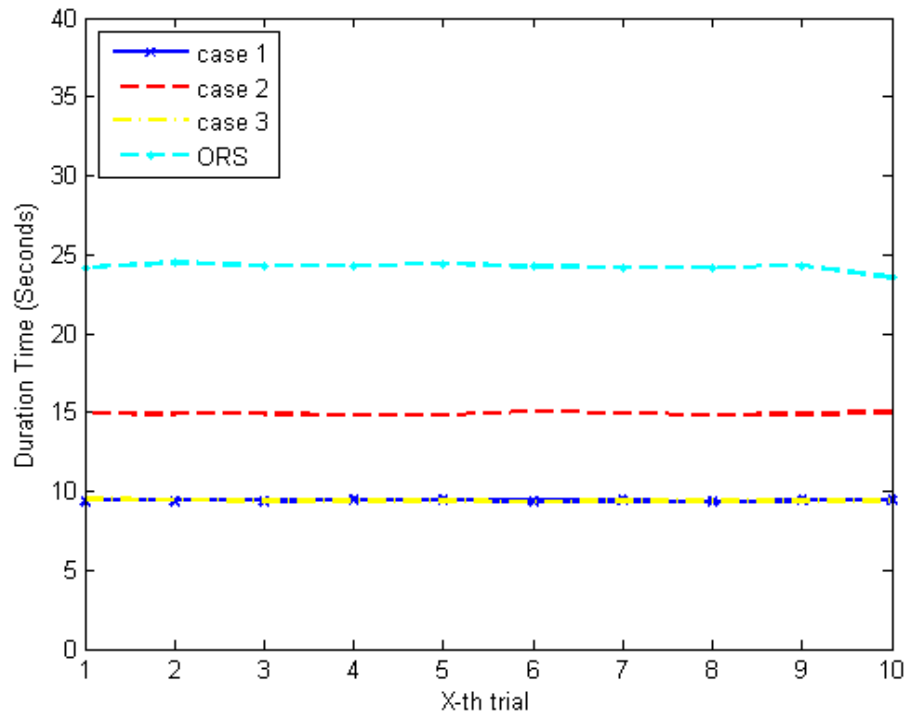


Figure 5.4: The line graph of duration time for running the test cases involved Shopping A.

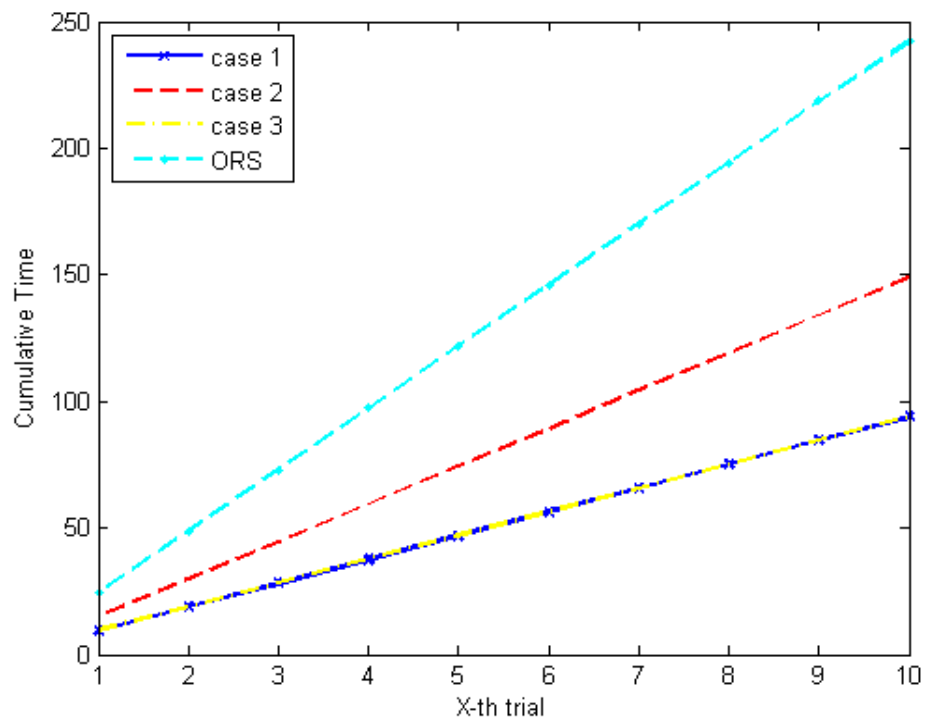


Figure 5.5: The cumulative time of running the test cases involved Shopping A.

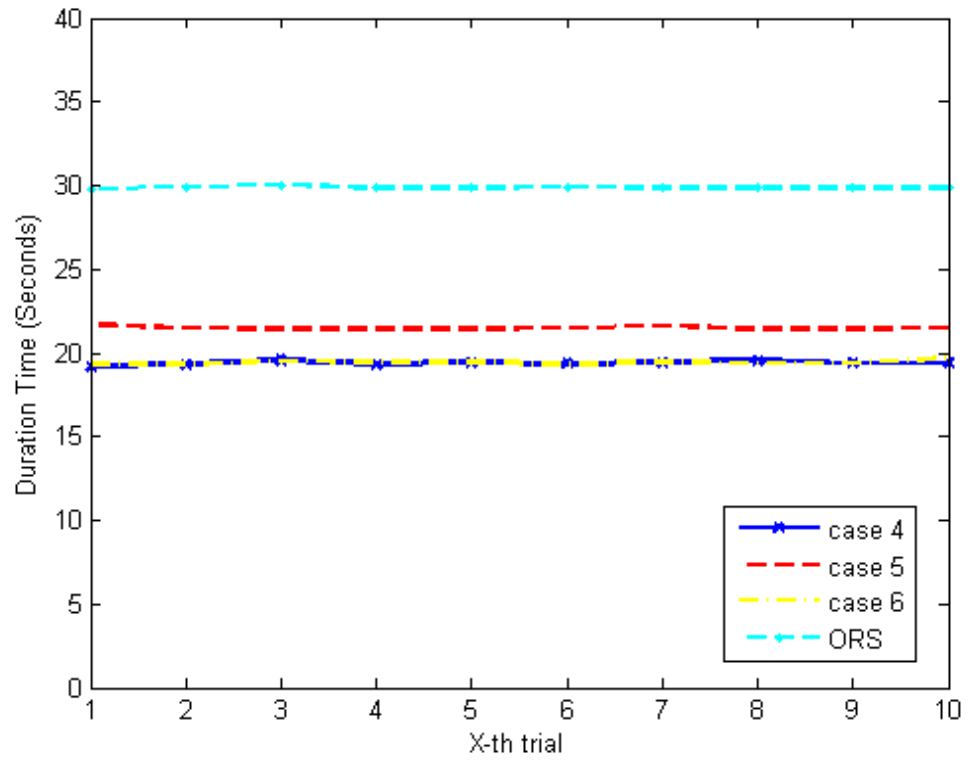


Figure 5.6: The line graph of duration time for running the test cases involved Shopping B.

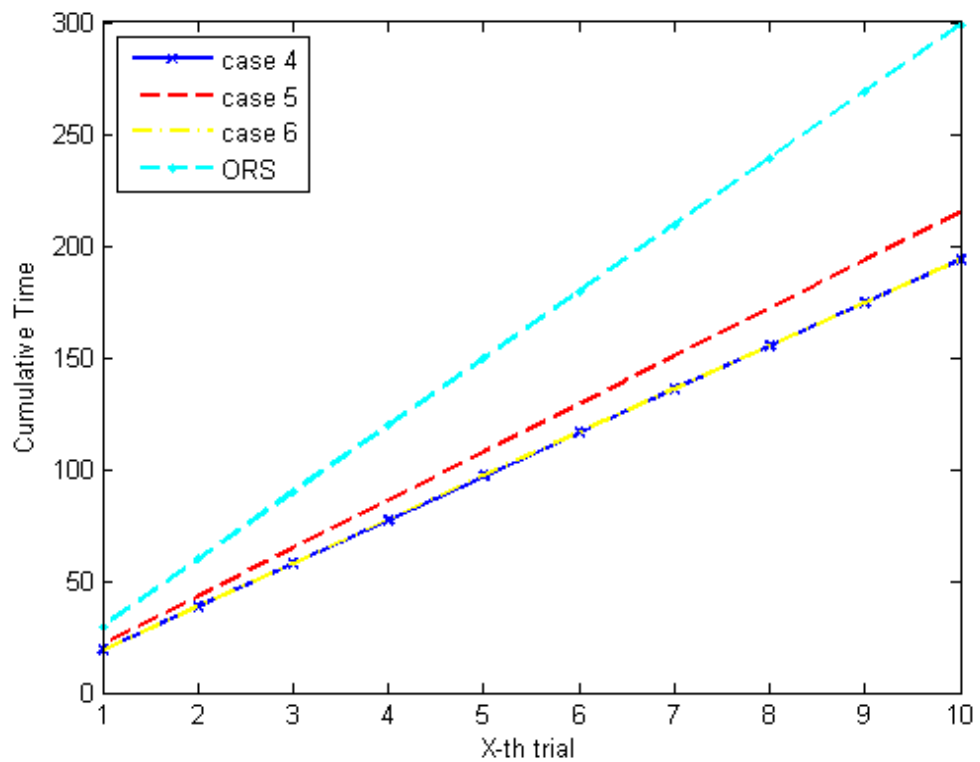


Figure 5.7: The cumulative time of running the test cases involved Shopping B.

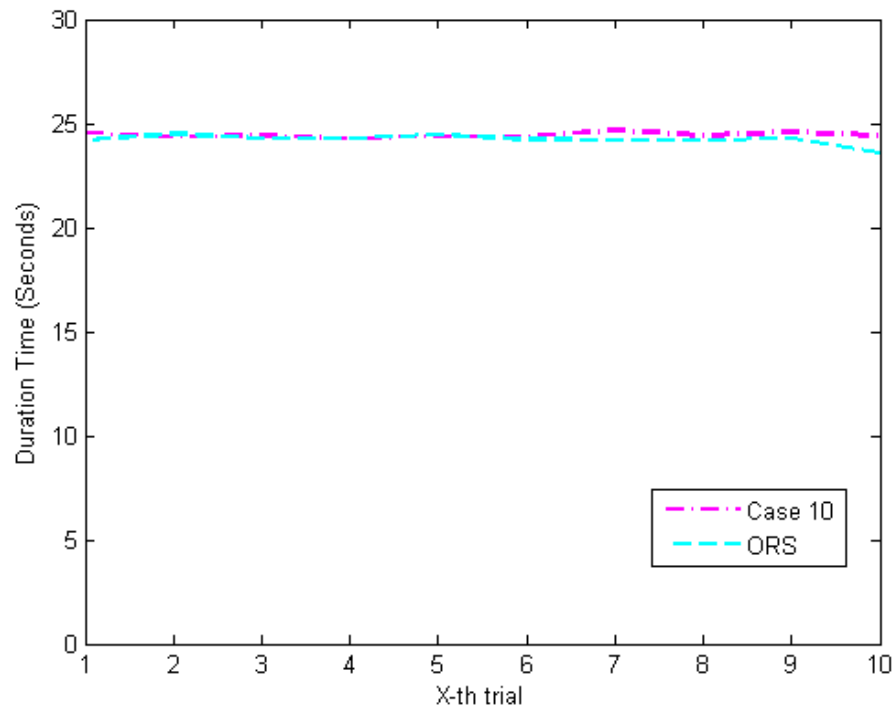


Figure 5.8: The line graph of duration time for running Shopping A to see how much time the ORS waste with bridge rule component.

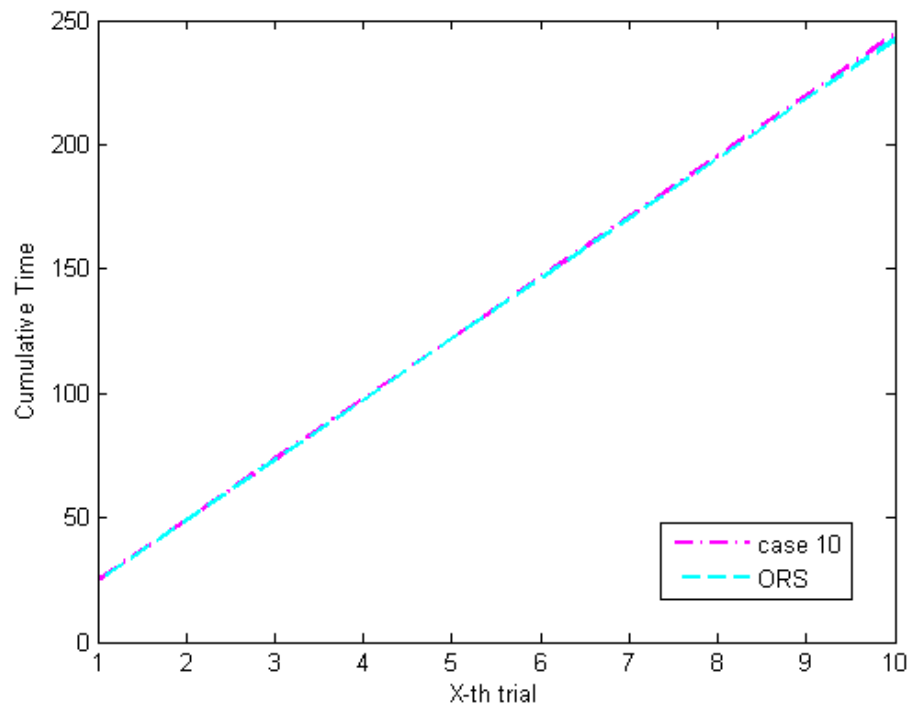


Figure 5.9: The cumulative time of running Shopping A to see how much time the ORS waste with the bridge rule component.

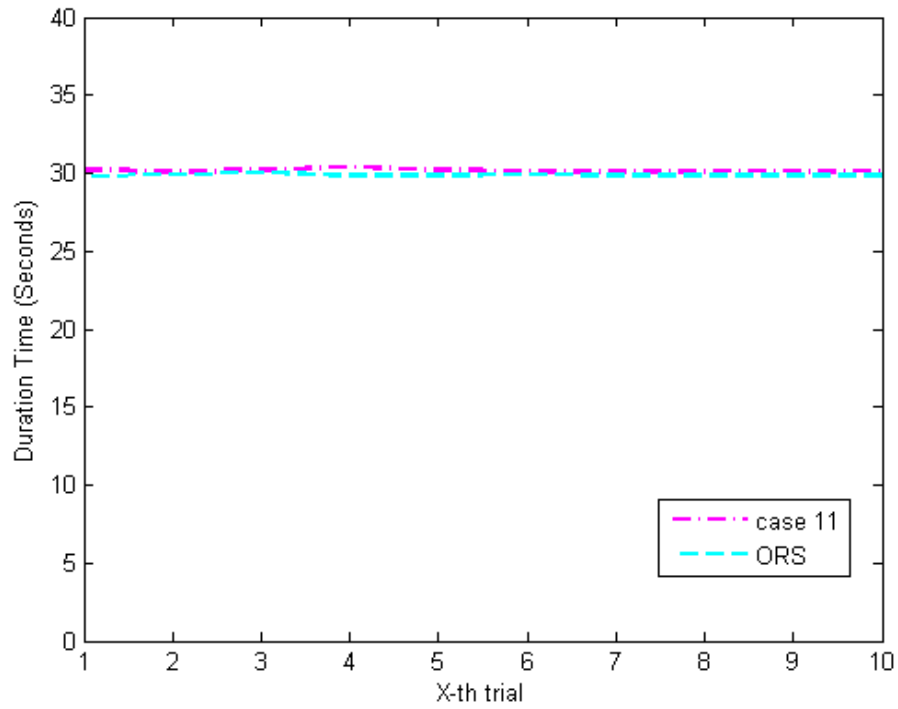


Figure 5.10: The line graph of duration time for running Shopping B to see how much time the ORS waste with bridge rule component.

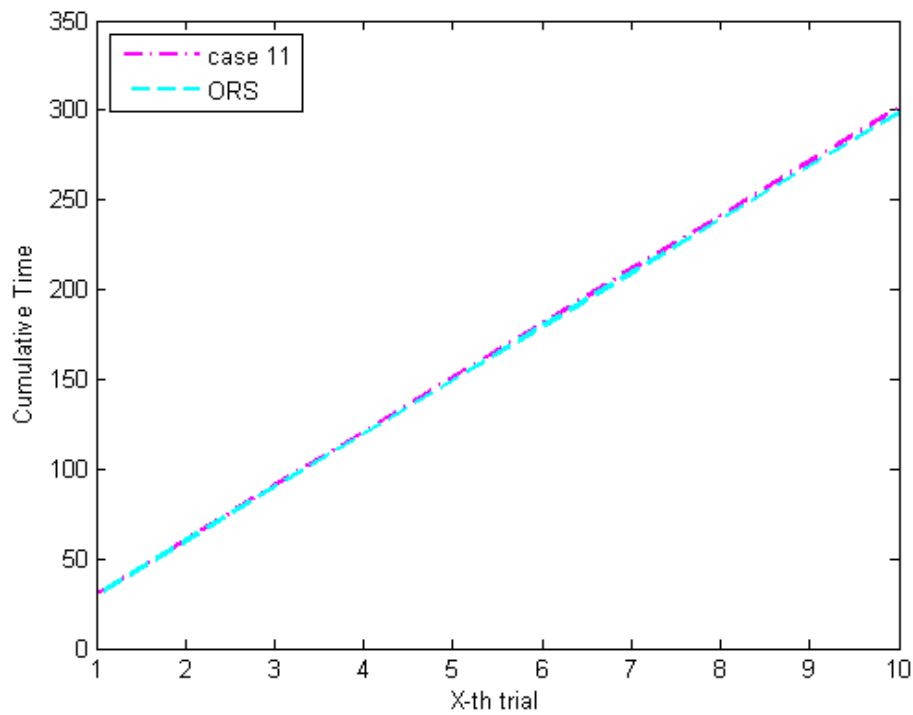


Figure 5.11: The cumulative time of running Shopping B to see how much time the ORS waste with the bridge rule component.

5.5 Analysis

After reported results, we found that the results were as we expected. Therefore, we will discuss and evaluate our bridge rule component compared with our expectation as we provided earlier in this chapter.

5.5.1 Evaluation on Effectiveness

The results in term of effectiveness are as we expect. All test cases that could find the relevant bridge rules contain less failure and fewer repairs compared with the ORS without bridge rule.

According to section 5.3.2, we explain three sets of bridge rules for testing across three shopping ontologies. In case 1, we tested Shopping A and Set A and the result was as we expect; that is no failure and no repair. The PA answered the question from SPA by using the information in bridge rules and the plan continued to achieve the goal smoothly. This is because Set A contains all information that fully matches with Shopping A. For other cases that involve Shopping A, the amount of failure is also less than the original ORS. In Case 2, there was one failure because Set B is intended to test how well the system works with the combination of using bridge rules and performing diagnosis. Instead of two failures from the original ORS, Case 2 only failed once which means that the PA found two answers in Set B and the other by performing diagnosis and repairing the PA's ontology. However, the result in Case 3 was no failure as we expected because Set C contains information that we need to capture the compound mismatch in Shopping C and also common mismatches in our three ontologies. Since Shopping A only comprises of the common mismatches, this bridge rule should be matched with the ontology.

In case 4, case 5 and case 6, the results were as we expected. Set A was created to fully match with Shopping A then it may not fully match with Shopping B. Set C also was created to capture the compound mismatch in Shopping C. Therefore, Case 4 and Case 6 could only find two answers instead of four in bridge rules and caused two failures and repairs during the execution. For Case 5, we test how well the extended ORS work with a combination of using bridge rules and performing

diagnosis. It showed that when the PA found the answer from bridge rules it then skipped the diagnosis and continued the plan until the plan succeeded or it could not satisfy a query then the PA called for diagnosis. After diagnosis and repair, the repair information was added into Set B. However, all these cases performed better than the original ORS that they contained less failures and fewer repairs.

For Shopping C, only the result in Case 9 was no failure as we expected. Since the limit of current ORS is compound mismatch, we try to figure how bridge rule could be useful with this type. Therefore, we assume that if we have two or more repair information that could link one to another then we will find the compound mismatch by linking them together and return the possible answer. Therefore, in Set C we created a set of bridge rules that can match with the common mismatch and two mismatches that can compose to the compound mismatch. The two mismatches stored in Set C are:

$$\textit{chooseThing}(\textit{Agent}, \textit{Book}) \rightarrow \textit{ChooseItem}(\textit{Agent}, \textit{Book})$$

$$\textit{ChooseItem}(\textit{Agent}, \textit{Book}) \rightarrow \textit{ChooseItem}(\textit{Agent}, \textit{Book}, \textit{Ref}).$$

Which we expect them to be linked as the new rule

$$\textit{chooseThing}(\textit{Agent}, \textit{Book}) \rightarrow \textit{ChooseItem}(\textit{Agent}, \textit{Book}, \textit{Ref})$$

This rule matched with Shopping C and the plan continued and accomplished the goal.

According to the results, we can conclude that with bridge rules the communication between the PA and SPAs leads to less fail and fewer repair compared to the original ORS. Therefore, the evaluation shows that the effectiveness of the extended ORS is better than the original ORS and the results match with our hypothesis.

5.5.2 Evaluation on Efficiency

As we expect, the efficiency with bridge rules is better than ORS without bridge rule. In Figure 5.4 and 5.5, the execution time of using bridge rules with Shopping A is twice as fast as the original ORS in Case 1 and Case 3. Case 2 is also faster but less than other cases because it also needs to perform the diagnosis and repair the ontology. For Shopping B, Case 4, 5 and 6 are also faster than the original as in Figure 5.6 and 5.7. However, they are not faster as Case 1 and Case 3 because bridge

rules did not fully match with the ontology then we need to perform diagnosis and repair the ontology. Although we cannot compare the running of Shopping C with the original, we can see that in Case 9 the average time is 9.384 seconds which is close to other cases. Then we infer that it took little time to run the system. As we can see for the graph, the variances are small because all the lines in cumulative graphs almost the straight line so we can conclude for the average results that the execution time of using bridge rules are faster than performing the diagnosis and repair.

Another case that could be considered is the time that ORS wastes searching bridge rules every time the PA cannot answer the question and the time that it wastes storing the repair information after performing the repair to the ontology. For both Shopping A and C, the execution time of looking up and storing bridge rules did not have any significant change to the original system as shown in Figure 5.8 – 5.11. There are almost the same lines if we see in the cumulative graph in Figure 5.9 and Figure 5.11. Thus, we can conclude that bridge rule component does not waste a lot of time to perform.

In terms of efficiency, we can conclude that with bridge rule component is efficient compared with the original as we can see from the results. Since we tested and evaluated in the small sets of bridge rules, if we use the large database, the result might be slow and we are aware of this problem. We will discuss the possible work for a large database in Chapter 6.

5.6 Related Works

As discussed in Chapter 2, we introduced the existing studies that related to our works. Since we have collected and evaluated our work, we can now discuss and compare our results with the related works. In this section, we provide the related work and compare it with our work.

One of the related works is versioning ontologies as explained in [21, 22]. We have mentioned that they track changes during the ontology evolution and create a version log to keep these changes. However, the work mostly focuses on the offline environment and requires full access to the ontology. The main purpose of version log is to check and maintain the consistency of the ontology after several changes. In

contrast, our project focuses on keeping and maintaining the repair information to be reused and reduce execution failures.

The definition of bridge rules are introduced and used in [3, 18] as we introduced in Chapter 2. Although their definition is similar to our bridge rules - that is the rule for mapping between two ontologies - their purpose is different. In [3], the C-OWL syntax is introduced which is composed of OWL syntax and bridge rules to give the semantics link to different ontologies. Therefore, it only restricts to OWL format and need to have full access to the ontologies. This is in contrast with ORS concepts. The work in [18] focuses on automated repair of the ontology that caused by the ontology mapping. It checks the inconsistency of the ontology using the bridge rules, which is not in our scope because the original ORS already performs this task while it repairs the ontology.

The most relevant work is on experience sharing in [13]. Hu extended ORS to enable experience sharing between again. The experiences consist of the repair information that previous successful repaired by ORS. The experiences can be shared with others agents, then agents can consume the experiences and perform repair using this experiences. He expected that with experience sharing it could perform more efficiency than the original ORS. He also tested the system with the compound mismatch to see whether it is possible to capture this type with the experiences or not. However, the evaluation result showed that it is unable to capture the compound mismatch and the execution time cannot indicate the efficient result. Moreover, experience sharing is consumed after the PA starts the diagnostic process, which means that after the action fails then it needs to replan and restart the new cycle again after repair. In contrast with our project, the PA can look up bridge rules before the action fails and the ontology is not changed by using bridge rules.

Chapter 6

Further Works

Despite the positive results, there are some issues that need to be considered and should be discussed as the extension of this project. In this chapter, we introduce possible further works that could be applied to this project.

Since our project is designed and implemented based on ORS, the bridge rule database was also a Prolog based database for more convenience. If we consider in a large scale environment which many types of mismatches could occur, a Prolog based database might not be suitable to maintain a bridge rule record. For further work, we could consider storing bridge rules in the database such as a relational database that is possible to access with any kinds of programming languages.

Another further work is bridge rule selection. Currently, we focus on the agent environment in ORS which is a small scale of the multiagent system. Therefore, the number of action failure and ontological mismatches is not high compared with the large scale environment. If the number of ontological mismatches increases, ORS will identify and repair them and then these repair information are recorded in bridge rule database. Although our new component can return the possible matches, the searching method could be improved by using statistical approaches. The idea of using statistics is because some bridge rules are used more often than the others. It is possible to keep track how many times these bridge rules are called. If we have a bridge rule database containing the same LHS terms but different RHS terms, it would be faster to select the rules that have higher frequency than the others. For example, there are two bridge rules in the database:

$money(Agent, Amount) \rightarrow Dollar(Agent, Amount)$ Frequency:3

$money(Agent, Amount) \rightarrow money(Agent, Currency, Amount)$ Frequency:6

Currently in searching method, these two rules are selected and both LHS terms and check in the PA's ontology and return one that is found. However, we could use statistics to select the most probable term if we keep track the frequency of each rule. In this example, we can notice that the second rule is used more than the first rule. This approach would be faster if we have a large number of bridge rules in the database then it can select the most probable rule to check in the PA's ontology instead of checking all the rules that match. Since we have a limit of time, this statistical approach is considered as the further expansion of the system.

In addition, we could apply the statistical approach to change the ontology. If we notice that this bridge rule is used often, it would be better to change the ontology back to the previous repair then the PA does not have to look up the bridge rule for this case. As the example earlier, we notice that the second rule is used 6 times. Therefore, we could consider modifying the PA's ontology to contain $money(Agent, Amount)$ rather than $money(Agent, Currenct, Amount)$. This may be reduce the number of times that the PA answer 'no' and look up bridge rules.

Chapter 7

Conclusion

In multiagent systems, agent communication is essential. Since the individual agent is developed by different people and contains different knowledge, the communication problems cannot be avoided. One of the problems is ontological mismatch, because each agent may represent their knowledge differently. During interaction, if agents cannot understand the knowledge from other agents, communication will fail. In this project, we investigated how to decrease execution failure and the number of ontological mismatches.

Our project is based on the ontology repair system which discussed in Chapter 2. ORS is a system which can identify and repair the ontology during agent execution which attempts to solve the problem of ontological mismatch in multiagent systems. We explored the limitation of ORS that it did not keep track the changes after repair was successfully made. Then we explained our hypothesis that the information is useful for further communication and can reduce the number of failure and repairs.

We introduced the concept of a bridge rule to store the repair information and explained our design and components in Chapter 4. The bridge rule composed of the term before repair and the term after repair, together with the rule, the repair type and repair information. We divided the component into two tasks: searching and storing. For searching, the PA could find relevant information to answer the question that it cannot answer. If the PA could not find a bridge rule, it will call diagnosis and repair the ontology. After repair succeeds, the repair information is extracted and sent to be stored in a bridge rule format.

We implemented this new component and conducted experiments to evaluate our extended ORS compared with the current ORS. The results showed that our new component is effective and efficient. It reduced the execution failure and captured the compound mismatches which are limitation in the current ORS. Using bridge rule is faster than performing the diagnosis and repair. Since we have a limit of time, we gave the possible further works to develop this system in Chapter 6.

Bibliography

- [1] F. Baader, I. Horrocks, and U. Sattler, "Description logics as ontology languages for the semantic web," in *Mechanizing Mathematical Reasoning*, ed: Springer Berlin Heidelberg, 2005, pp. 228–248.
- [2] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific american*, vol. 284, pp. 28–37, 2001.
- [3] P. Bouquet, F. Giunchiglia, F. Van Harmelen, L. Serafini, and H. Stuckenschmidt, "Contextualizing ontologies," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, pp. 325–343, 2004.
- [4] E. H. Durfee, "Distributed Problem Solving and Planning," in *Multi-Agent Systems and Applications*. vol. 2086, M. Luck, *et al.*, Eds., ed Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 118-149.
- [5] E. H. Durfee and J. S. Rosenschein, "Distributed problem solving and multi-agent systems: Comparisons and examples," in *Proceedings of the 13th International DAI Workshop*, ed. Seattle, Wash., 1994.
- [6] A. Farquhar, R. Fikes, and J. Rice, "The ontolingua server: A tool for collaborative ontology construction," *International Journal of Human-Computers Studies*, vol. 46, pp. 707–727, 1997.
- [7] M. R. Genesereth and R. E. Fikes, "Knowledge Interchange Format Version 3.0 Reference Manual, Logic-92-1," *Computer Science Department, Stanford University*, 1992.
- [8] T. R. Gruber, "Ontolingua: A mechanism to support portable ontologies, Knowledge Systems Laboratory Technical Report KSL-91-66," *Stanford University*, 1992.
- [9] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge acquisition*, vol. 5, pp. 199–199, 1993.

- [10] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *International Journal of Human Computer Studies*, vol. 43, pp. 907–928, 1995.
- [11] N. Guarino, "Formal ontology in information systems," in *Proceedings of FOIS'98*, ed. Trento, Italy: IOS Press, 1998, pp. 3-15.
- [12] A. Hameed, A. Preece, and D. Sleeman, "Ontology reconciliation," in *Handbook on ontologies*, S. Staab and R. Studer, Eds., ed: Springer Verlag, Germany, 2004, pp. 231–250.
- [13] K. Hu, "Experience Sharing with the Ontology Repair System," Master of Science, School of Informatics, University of Edinburgh, Edinburgh, 2010.
- [14] Y. Kalfoglou, "Exploring ontologies," *Handbook of Software Engineering and Knowledge Engineering: vol. 1: Fundamentals*, pp. 863–887, 2001.
- [15] Y. Kalfoglou and M. Schorlemmer, "Ontology mapping: the state of the art," *The knowledge engineering review*, vol. 18, pp. 1–31, 2003.
- [16] F. McNeill, "Dynamic Ontology Refinement," PhD Thesis, University of Edinburgh, 2006.
- [17] F. McNeill and A. Bundy, "Dynamic, automatic, first-order ontology repair by diagnosis of failed plan execution," *International Journal on Semantic Web and information systems*, vol. 3, pp. 1–35, 2007.
- [18] C. Meilicke, H. Stuckenschmidt, and A. Tamin, "Repairing ontology mappings," in *Proceedings of the National Conference on Artificial Intelligence*, 2007, p. 1408.
- [19] N. F. Noy and M. A. Musen, "PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment," in *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, 2000, pp. 450-455.
- [20] N. F. Noy and M. A. Musen, "Ontology versioning in an ontology management framework," *Intelligent Systems, IEEE*, vol. 19, pp. 6-13, 2004.
- [21] P. Plessers and O. De Troyer, "Ontology Change Detection Using a Version Log," *IN PROCEEDING OF THE 4TH INTERNATIONAL SEMANTIC WEB CONFERENCE*, pp. 578--592, 2005.

- [22] P. Plessers, O. De Troyer, and S. Casteleyn, "Understanding ontology evolution: A change detection approach," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, pp. 39-49, 2007.
- [23] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*: Pearson Education, 2010.
- [24] L. Serafini, A. Borgida, and A. Taminin, "Aspects of distributed and modular ontology reasoning," presented at the International Joint Conference on Artificial Intelligence, 2005.
- [25] B. Smith, "Ontology," in *The Blackwell Guide to the Philosophy of Computing and Information*, L. Floridi, Ed., ed Malden, MA, Oxford: Blackwell, 2004, pp. 155–166.
- [26] M. Wooldridge, *An Introduction to MultiAgent Systems*: John Wiley & Sons, 2009.
- [27] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *Knowledge engineering review*, vol. 10, pp. 115–152, 1995.

Appendix A

The Shopping Ontology

A.1 Shopping A

```
(In-Package "ONTOLINGUA-USER")
;;; Written by user Fionam from session "learners" owned by
group UNIVERSE
;;; Date: Nov 17, 2004 09:35
;;; Definitions: 54
(Define-Ontology
  Sem-Web-Lucas
  (Frame-Ontology)
  "Not supplied yet."
  :Io-Package
  "ONTOLINGUA-USER")
(In-Ontology (Quote Shopping))
;;; Item
(Define-Class Item (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Group
(Define-Class Group (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Shopping-Group
(Define-Class Shopping-Group (?X) "Not supplied yet." :Def
(And (Group ?X)))
;;; Academic-Group
(Define-Class Academic-Group (?X) "Not supplied yet." :Def
(And (Group ?X)))
;;; Agent
(Define-Class Agent (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Book
(Define-Class Book (?X) "Not supplied yet." :Def (And (Item
?X)))
;;; Buy
(Define-Axiom Buy "Not supplied yet." := (=> (And (In-Basket
?Agent ?Item Pseudo-Var ?Sit1) (Money ?Agent ?Amount ?Sit1)
```

```

(Cost ?Item ?Price ?Sit1) (< ?Price ?Amount)) (And (Has ?Agent
?Item ?Sit2) (= ?Newamount (- ?Amount ?Price)) (Money ?Agent
?Newamount ?Sit2) (Not (Money ?Agent ?Amount ?Sit2))))
;;; Has
(Define-Relation Has (?Agent ?Item ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Item ?Item) (Sit-Var
?Situation)))
;;; In-Basket
(Define-Function In-Basket (?Agent-0 ?Item ?Situation) :->
?Value "Not supplied yet." :Def (And (Agent ?Agent-0) (Item
?Item) (Confirmation-Number ?Value) (Sit-Var ?Situation)))
;;; Choose-Item
(Define-Relation Choose-Item (?Agent ?Item ?Situation) "Not
supplied yet." :Def (And (Agent ?Agent) (Item ?Item) (Sit-Var
?Situation)))
;;; Choose-Thing
(Define-Relation Choose-Thing (?Agent ?Item ?Situation) "Not
supplied yet." :Def (And (Agent ?Agent) (Item ?Item) (Sit-Var
?Situation)))
;;; Registered-Member
(Define-Relation Registered-Member (?Agent ?Group ?Situation)
"Not supplied yet." :Def (And (Agent ?Agent) (Group ?Group)
(Sit-Var ?Situation)))
;;; Cost
(Define-Function Cost (?Item ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Item ?Item) (Number ?Value) (Sit-Var
?Situation)))
;;; Money
(Define-Function Money (?Agent-0 ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Agent ?Agent-0) (Number ?Value)
(Sit-Var ?Situation)))
;;; Put-In-Basket
(Define-Axiom Put-In-Basket "Not supplied yet." := (=> (And
(Choose-Item ?Agent ?Item ?Sit1) (Registered-Member ?Agent
?Group) (Group ?Group)) (And (In-Basket ?Agent ?Item Pseudo-
Var ?Sit2))))
;;; Join-Group
(Define-Axiom Join-Group "Not supplied yet." := (=> (And
(Group ?Group)) (And (Registered-Member ?Agent ?Group))))
;;; Our-Mutual-Friend
(Define-Frame Our-Mutual-Friend :Own-Slots ((Documentation
"Not supplied yet.") (Instance-Of Book)) :Axioms ((Cost Our-
Mutual-Friend 9 [(Start)]))
;;; Book-Shop-Group
(Define-Individual Book-Shop-Group (Shopping-Group) "Not
supplied yet.")
;;; Ai-Group
(Define-Individual Ai-Group (Academic-Group) "Not supplied
yet.")
;;; Shopping-Agent
(Define-Frame Shopping-Agent :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent) (Registered-Member Ai-

```

```

Group [(Start)]) (Choose-Item Our-Mutual-Friend [(Start)]))
: Axioms ((Money Shopping-Agent 100 [(Start)]))
;;; Currency
(Define-Class Currency (?X) "Not supplied yet." :Def (And
(Thing ?X)))
;;; Dollars
(Define-Individual Dollars (Currency) "Not supplied yet.")
;;; Sterling
(Define-Function Sterling (?Agent-0 ?Situation) :-> ?Value
"Not supplied yet." :Def (And (Agent ?Agent-0) (Number ?Value)
(Sit-Var ?Situation)))
;;; Confirmation-Number
(Define-Class Confirmation-Number (?X) "Not supplied yet."
:Def (And (Thing ?X)))
;;; Pseudo-Var
(Define-Individual Pseudo-Var (Confirmation-Number) "Not
supplied yet.")
;;; Sit-Var
(Define-Class Sit-Var (?X) "Not supplied yet." :Def (And
(Thing ?X)))
;;; Start
(Define-Individual Start (Sit-Var) "Not supplied yet.")

```

A.2 Shopping B

```

(In-Package "ONTOLINGUA-USER")
;;; Written by user Fionam from session "learners" owned by
group UNIVERSE
;;; Date: Nov 17, 2004 09:35
;;; Definitions: 54
(Define-Ontology
  Sem-Web-Lucas
  (Frame-Ontology)
  "Not supplied yet."
  :Io-Package
  "ONTOLINGUA-USER")
(In-Ontology (Quote Shopping))
;;; Item
(Define-Class Item (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Group
(Define-Class Group (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Shopping-Group
(Define-Class Shopping-Group (?X) "Not supplied yet." :Def
(And (Group ?X)))
;;; Academic-Group
(Define-Class Academic-Group (?X) "Not supplied yet." :Def
(And (Group ?X)))
;;; Agent
(Define-Class Agent (?X) "Not supplied yet." :Def (And (Thing
?X)))

```

```

;;; Book
(Define-Class Book (?X) "Not supplied yet." :Def (And (Item
?X)))
;;; Buy
(Define-Axiom Buy "Not supplied yet." := (=> (And (In-Basket
?Agent ?Item Pseudo-Var ?Sit1) (Money ?Agent ?Amount ?Sit1)
(Cost ?Item ?Price ?Sit1) (< ?Price ?Amount)) (And (Has ?Agent
?Item ?Sit2) (= ?Newamount (- ?Amount ?Price)) (Money ?Agent
?Newamount ?Sit2) (Not (Money ?Agent ?Amount ?Sit2)))))
;;; Has
(Define-Relation Has (?Agent ?Item ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Item ?Item) (Sit-Var
?Situation)))
;;; In-Basket
(Define-Function In-Basket (?Agent-0 ?Item ?Confirmation-
Number ?Situation) :-> ?Value "Not supplied yet." :Def (And
(Agent ?Agent-0) (Item ?Item) (Confirmation-Number ?Value)
(Sit-Var ?Situation)))
;;; Choose-Item
(Define-Relation Choose-Item (?Agent ?Item ?Situation) "Not
supplied yet." :Def (And (Agent ?Agent) (Item ?Item) (Sit-Var
?Situation)))
;;; Interest
(Define-Relation Interest (?Agent ?Group ?Situation) "Not
supplied yet." :Def (And (Agent ?Agent) (Group ?Group) (Sit-
Var ?Situation)))
;;; Registered-Member
(Define-Relation Registered-Member (?Agent ?Group ?Situation)
"Not supplied yet." :Def (And (Agent ?Agent) (Group ?Group)
(Sit-Var ?Situation)))
;;; Cost
(Define-Function Cost (?Item ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Item ?Item) (Number ?Value) (Sit-Var
?Situation)))
;;; Money
(Define-Function Money (?Agent-0 ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Agent ?Agent-0) (Number ?Value)
(Sit-Var ?Situation)))
;;; Put-In-Basket
(Define-Axiom Put-In-Basket "Not supplied yet." := (=> (And
(Choose-Item ?Agent ?Item ?Sit1) (Registered-Member ?Agent
?Group ?Sit1) (Group ?Group)) (And (In-Basket ?Agent ?Item
Pseudo-Var ?Sit2))))
;;; Join-Group
(Define-Axiom Join-Group "Not supplied yet." := (=> (And
(Interest ?Agent ?Group ?Sit1) (Group ?Group)) (And
(Registered-Member ?Agent ?Group ?Sit2))))
;;; Our-Mutual-Friend
(Define-Frame Our-Mutual-Friend :Own-Slots ((Documentation
"Not supplied yet.") (Instance-Of Book)) :Axioms ((Cost Our-
Mutual-Friend 9 [(Start)])))
;;; Book-Shop-Group

```

```

(Define-Individual Book-Shop-Group (Shopping-Group) "Not
supplied yet.")
;;; Ai-Group
(Define-Individual Ai-Group (Academic-Group) "Not supplied
yet.")
;;; 123
(Define-Individual 123 (Confirmation-Number) "Not supplied
yet.")
;;; Ai-Group
(Define-Individual MyRef (Confirmation-Number) "Not supplied
yet.")
;;; Shopping-Agent
(Define-Frame Shopping-Agent :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent) (Registered-Member Ai-
Group [(Start)]) (Choose-Item Our-Mutual-Friend [(Start)])
(Interest Book-Shop-Group [(Start)])) :Axioms ((Money
Shopping-Agent 100 [(Start)]))
;;; Currency
(Define-Class Currency (?X) "Not supplied yet." :Def (And
(Thing ?X)))
;;; Dollars
(Define-Individual Dollars (Currency) "Not supplied yet.")
;;; Sterling
(Define-Function Sterling (?Agent-0 ?Situation) :-> ?Value
"Not supplied yet." :Def (And (Agent ?Agent-0) (Number ?Value)
(Sit-Var ?Situation)))
;;; Confirmation-Number
(Define-Class Confirmation-Number (?X) "Not supplied yet."
:Def (And (Thing ?X)))
;;; Pseudo-Var
(Define-Individual Pseudo-Var (Confirmation-Number) "Not
supplied yet.")
;;; Sit-Var
(Define-Class Sit-Var (?X) "Not supplied yet." :Def (And
(Thing ?X)))
;;; Start
(Define-Individual Start (Sit-Var) "Not supplied yet.")

```

A.3 Shopping C

```

(In-Package "ONTOLINGUA-USER")
;;; Written by user Fionam from session "learners" owned by
group UNIVERSE
;;; Date: Nov 17, 2004 09:35
;;; Definitions: 54
(Define-Ontology
  Sem-Web-Lucas
  (Frame-Ontology)
  "Not supplied yet."
  :Io-Package
  "ONTOLINGUA-USER")
(In-Ontology (Quote Shopping))

```

```

;;; Item
(Define-Class Item (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Group
(Define-Class Group (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Shopping-Group
(Define-Class Shopping-Group (?X) "Not supplied yet." :Def
(And (Group ?X)))
;;; Academic-Group
(Define-Class Academic-Group (?X) "Not supplied yet." :Def
(And (Group ?X)))
;;; Agent
(Define-Class Agent (?X) "Not supplied yet." :Def (And (Thing
?X)))
;;; Book
(Define-Class Book (?X) "Not supplied yet." :Def (And (Item
?X)))
;;; Buy
(Define-Axiom Buy "Not supplied yet." := (=> (And (In-Basket
?Agent ?Item Pseudo-Var ?Sit1) (Money ?Agent ?Amount ?Sit1)
(Cost ?Item ?Price ?Sit1) (< ?Price ?Amount)) (And (Has ?Agent
?Item ?Sit2) (= ?Newamount (- ?Amount ?Price)) (Money ?Agent
?Newamount ?Sit2) (Not (Money ?Agent ?Amount ?Sit2))))
;;; Has
(Define-Relation Has (?Agent ?Item ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Item ?Item) (Sit-Var
?Situation)))
;;; In-Basket
(Define-Function In-Basket (?Agent-0 ?Item ?Confirmation-
Number ?Situation) :-> ?Value "Not supplied yet." :Def (And
(Agent ?Agent-0) (Item ?Item) (Confirmation-Number ?Value)
(Sit-Var ?Situation)))
;;; Choose-Item
(Define-Relation Choose-Item (?Agent ?Item ?Ref ?Situation)
"Not supplied yet." :Def (And (Agent ?Agent) (Item ?Item)
(Confirmation-Number ?Ref) (Sit-Var ?Situation)))
;;; Interest
(Define-Relation Interest (?Agent ?Group ?Situation) "Not
supplied yet." :Def (And (Agent ?Agent) (Group ?Group) (Sit-
Var ?Situation)))
;;; Registered-Member
(Define-Relation Registered-Member (?Agent ?Group ?Situation)
"Not supplied yet." :Def (And (Agent ?Agent) (Group ?Group)
(Sit-Var ?Situation)))
;;; Cost
(Define-Function Cost (?Item ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Item ?Item) (Number ?Value) (Sit-Var
?Situation)))
;;; Money
(Define-Function Money (?Agent-0 ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Agent ?Agent-0) (Number ?Value)
(Sit-Var ?Situation)))

```

```

;;; Put-In-Basket
(Define-Axiom Put-In-Basket "Not supplied yet." := (=> (And
(Choose-Item ?Agent ?Item ?Ref ?Sit1) (Registered-Member
?Agent ?Group ?Sit1) (Group ?Group)) (And (In-Basket ?Agent
?Item Pseudo-Var ?Sit2))))
;;; Join-Group
(Define-Axiom Join-Group "Not supplied yet." := (=> (And
(Interest ?Agent ?Group ?Sit1) (Group ?Group)) (And
(Registered-Member ?Agent ?Group ?Sit2))))
;;; Our-Mutual-Friend
(Define-Frame Our-Mutual-Friend :Own-Slots ((Documentation
"Not supplied yet.") (Instance-Of Book)) :Axioms ((Cost Our-
Mutual-Friend 9 [(Start)]))
;;; Book-Shop-Group
(Define-Individual Book-Shop-Group (Shopping-Group) "Not
supplied yet.")
;;; Ai-Group
(Define-Individual Ai-Group (Academic-Group) "Not supplied
yet.")
;;; 123
(Define-Individual 123 (Confirmation-Number) "Not supplied
yet.")
;;; Ai-Group
(Define-Individual MyRef (Confirmation-Number) "Not supplied
yet.")
;;; Shopping-Agent
(Define-Frame Shopping-Agent :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent) (Registered-Member Ai-
Group [(Start)]) (Choose-Item Our-Mutual-Friend 123 [(Start)])
(Interest Book-Shop-Group [(Start)])) :Axioms ((Money
Shopping-Agent 100 [(Start)]))
;;; Currency
(Define-Class Currency (?X) "Not supplied yet." :Def (And
(Thing ?X)))
;;; Dollars
(Define-Individual Dollars (Currency) "Not supplied yet.")
;;; Sterling
(Define-Function Sterling (?Agent-0 ?Situation) :-> ?Value
"Not supplied yet." :Def (And (Agent ?Agent-0) (Number ?Value)
(Sit-Var ?Situation)))
;;; Confirmation-Number
(Define-Class Confirmation-Number (?X) "Not supplied yet."
:Def (And (Thing ?X)))
;;; Pseudo-Var
(Define-Individual Pseudo-Var (Confirmation-Number) "Not
supplied yet.")
;;; Sit-Var
(Define-Class Sit-Var (?X) "Not supplied yet." :Def (And
(Thing ?X)))
;;; Start
(Define-Individual Start (Sit-Var) "Not supplied yet.")

```

Appendix B

The Stored Bridge Rules

B.1 Set A

```
bridgeRule([original,update],[repairType,[repairInfo]]).
```

```
bridgeRule([money(shoppingAgent,dollars,100),money(shoppingAgent,100)],[propositionalA,[money,3,[agent,currency,uninstantiated],currency,2]]).
```

```
bridgeRule([chooseThing(shoppingAgent,ourMutualFriend),chooseItem(shoppingAgent,ourMutualFriend)],[predicateAA,[chooseThing,2,[agent,book]]]).
```

B.2 Set B

```
bridgeRule([original,update],[repairType,[repairInfo]]).
```

```
bridgeRule([interest(bookShopGroup,shoppingAgent),interest(shoppingAgent,bookShopGroup)],[switchArgs,[interest,2,[shoppingGroup,agent],[agent,shoppingGroup],[shoppingGroup,agent]]]).
```

```
bridgeRule([chooseThing(shoppingAgent,ourMutualFriend),chooseThing(shoppingAgent,titchmarshMemoirs)],[domainA,[chooseThing,2,[agent,book],gardeningBook,subclass(gardeningBook,book)]]).
```

```
bridgeRule([money(shoppingAgent,dollars,100),money(shoppingAgent,100)],[propositionalA,[money,3,[agent,currency,uninstantiated],currency,2]]).
```

B.3 Set C

```
bridgeRule([original,update],[repairType,[repairInfo]]).
```

```
bridgeRule([money(shoppingAgent,dollars,100),money(shoppingAgent,100)],  
[propositionalA,[money,3,[agent,currency,uninstantiated],currency,2]]).
```

```
bridgeRule([chooseThing(shoppingAgent,ourMutualFriend),chooseItem(shoppingAgent,ourMutualFriend)],  
[predicateAA,[chooseThing,2,[agent,book]]]).
```

```
bridgeRule([chooseItem(shoppingAgent,ourMutualFriend),chooseItem(shoppingAgent,ourMutualFriend,123)],  
[propositionalAA,[chooseItem,2,[agent,book],number,3]]).
```

Appendix C

The Output for the main test cases

C.1 The Output from Case 1

```
| ?- start.
```

```
Consulting the PLANNER...
Consulting the ONTOLOGY UPDATER...
Consulting ORS...
```

```
Goal is:      has(shoppingAgent,ourMutualFriend)
```

```
Translating ...
```

```
Need to find a plan ...
```

```
This is the plan:
```

```
[putInBasket(shoppingAgent,aiGroup,ourMutualFriend),buy(shoppi
ngAgent,ourMutualFriend)]
```

```
Executing the plan ...
```

```
I'm going to ask putInBasketAgent to perform
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) for me
putInBasketAgent asked me class(aiGroup,group)
```

```
I told putInBasketAgent class(aiGroup,group)
putInBasketAgent asked me
registeredMember(shoppingAgent,aiGroup)
I told putInBasketAgent
registeredMember(shoppingAgent,aiGroup)
putInBasketAgent asked me
chooseThing(shoppingAgent,ourMutualFriend)
```

Looking up a bridge rule...

Predicate found in a bridge rule : chooseThing

An unknown term has the same arity as in a bridge rule : 2

An unknown term has the same argument classes: [agent,book]

RHS term in a bridge rule :
chooseItem(shoppingAgent,ourMutualFriend)

Checking the PA's ontology...

Found an unknown term in a bridge rule
:chooseThing(shoppingAgent,ourMutualFriend)

```
I told putInBasketAgent
chooseThing(shoppingAgent,ourMutualFriend)
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) performed
successfully
I'm going to ask buyAgent to perform
buy(shoppingAgent,ourMutualFriend) for me
buyAgent asked me money(shoppingAgent,dollars,_381377)
```

Looking up a bridge rule...

Predicate found in a bridge rule : money

An unknown term has the same arity as in a bridge rule : 3

An unknown term has the same argument classes:
[agent,currency,uninstantiated]

RHS term in a bridge rule : money(shoppingAgent,100)

Checking the PA's ontology...

Found an unknown term in a bridge rule
:money(shoppingAgent,dollars,100)

```
I told buyAgent money(shoppingAgent,dollars,100)
buyAgent asked me
inBasket(shoppingAgent,ourMutualFriend,_385228)
I told buyAgent inBasket(shoppingAgent,ourMutualFriend,47899)
buy(shoppingAgent,ourMutualFriend) performed successfully
The KIF ontology has been updated
```

The plan is completed

The following actions have been performed:
 [buy(shoppingAgent,ourMutualFriend),putInBasket(shoppingAgent,
 aiGroup,ourMutualFriend),start]

The following repairs have been performed:
 []

To terminate the program, type "t."

C.2 The Output from Case 5

```
| ?- start.
```

```
Consulting the PLANNER...
Consulting the ONTOLOGY UPDATER...
Consulting ORS...
```

```
|-----|
| Goal is: | has(shoppingAgent,ourMutualFriend)
|-----|
```

Translating ...

Need to find a plan ...

This is the plan:
 [putInBasket(shoppingAgent,aiGroup,ourMutualFriend),buy(shoppi
 ngAgent,ourMutualFriend)]

Executing the plan ...

I'm going to ask putInBasketAgent to perform
 putInBasket(shoppingAgent,aiGroup,ourMutualFriend) for me
 putInBasketAgent asked me class(aiGroup,shoppingGroup)

Looking up a bridge rule...

Not found in a bridge rule

I told putInBasketAgent no
 The KIF ontology has been updated

putInBasket(shoppingAgent,aiGroup,ourMutualFriend) failed. I
 am asking ORS for a diagnosis.

I am requesting a diagnosis ...I received a query about
 class(aiGroup,shoppingGroup), which I was not expecting to be
 asked about.

class(aiGroup,shoppingGroup) has the same name as the
 precondition class(aiGroup,group)

They also have the same arity (2)

My object is of the wrong class.

putInBasketAgent expected an object of class shoppingGroup
 whereas I thought that an object of class group would be
 acceptable.

group is a subclass of shoppingGroup so I need to be more
 specific about this object

*** I must check if the repair can be performed.

*** I am performing the repair

Saving a bridge rule...

successfully saved a brigde rule.

ORS proposed the following diagnosis repair:

[group,shoppingGroup,[precondAA,class]]

Goal is:	has(shoppingAgent,ourMutualFriend)
----------	------------------------------------

Translating ...

Need to find a plan ...

This is the plan:

```
[joinGroup(shoppingAgent,bookShopGroup),putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend),buy(shoppingAgent,ourMutualFriend)]
```

Executing the plan ...

```
I'm going to ask joinGroupAgentTwo to perform
joinGroup(shoppingAgent,bookShopGroup) for me
joinGroupAgentTwo asked me
interest(bookShopGroup,shoppingAgent)
```

Looking up a bridge rule...

Predicate found in a bridge rule : interest

An unknown term has the same arity as in a bridge rule : 2

An unknown term has the same argument classes:
[shoppingGroup,agent]

RHS term in a bridge rule :
interest(shoppingAgent,bookShopGroup)

Checking the PA's ontology...

```
Found an unknown term in a bridge rule
:interest(bookShopGroup,shoppingAgent)
I told joinGroupAgentTwo interest(bookShopGroup,shoppingAgent)
joinGroup(shoppingAgent,bookShopGroup) performed successfully
I'm going to ask putInBasketAgent to perform
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend) for
me
putInBasketAgent asked me class(bookShopGroup,shoppingGroup)
I told putInBasketAgent class(bookShopGroup,shoppingGroup)
putInBasketAgent asked me
registeredMember(shoppingAgent,bookShopGroup)
I told putInBasketAgent
registeredMember(shoppingAgent,bookShopGroup)
putInBasketAgent asked me
chooseThing(shoppingAgent,ourMutualFriend)
```

Looking up a bridge rule...

Predicate found in a bridge rule : chooseThing

An unknown term has the same arity as in a bridge rule : 2

An unknown term has the same argument classes: [agent,book]

RHS term in a bridge rule :
chooseThing(shoppingAgent,titchmarshMemoirs)

Checking the PA's ontology...

RHS term is not found in the ontology

Checking the link between each bridge rule...

No link was found

I told putInBasketAgent no
The KIF ontology has been updated

putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend)
failed. I am asking ORS for a diagnosis.

I am requesting a diagnosis ...I received a query about
chooseThing(shoppingAgent,ourMutualFriend), which I was not
expecting to be asked about.

No preconditions have the same name as chooseThing

DIAGNOSIS: Predicate anti-abstraction
chooseThing is a subtype of chooseItem

*** I must check if the repair can be performed.
*** I am performing the repair

I knew the type
The appropriate repair has been performed

Saving a bridge rule...

successfully saved a brigde rule.

ORS proposed the following diagnosis repair:
[chooseItem(shoppingAgent,ourMutualFriend),chooseThing(shoppin
gAgent,ourMutualFriend),[predicateAA,[chooseThing,chooseItem]]
]

Goal is:	has(shoppingAgent,ourMutualFriend)
----------	------------------------------------

Translating ...

Need to find a plan ...

This is the plan:

```
[putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend),buy(
shoppingAgent,ourMutualFriend)]
```

Executing the plan ...

```
I'm going to ask putInBasketAgent to perform
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend) for
me
putInBasketAgent asked me class(bookShopGroup,shoppingGroup)
I told putInBasketAgent class(bookShopGroup,shoppingGroup)
putInBasketAgent asked me
registeredMember(shoppingAgent,bookShopGroup)
I told putInBasketAgent
registeredMember(shoppingAgent,bookShopGroup)
putInBasketAgent asked me
chooseThing(shoppingAgent,ourMutualFriend)
I told putInBasketAgent
chooseThing(shoppingAgent,ourMutualFriend)
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend)
performed successfully
I'm going to ask buyAgent to perform
buy(shoppingAgent,ourMutualFriend) for me
buyAgent asked me money(shoppingAgent,dollars,_2165436)
```

Looking up a bridge rule...

Predicate found in a bridge rule : money

An unknown term has the same arity as in a bridge rule : 3

An unknown term has the same argument classes:

```
[agent,currency,uninstantiated]
```

RHS term in a bridge rule : money(shoppingAgent,100)

Checking the PA's ontology...

Found an unknown term in a bridge rule
 :money(shoppingAgent,dollars,100)

I told buyAgent money(shoppingAgent,dollars,100)
 buyAgent asked me
 inBasket(shoppingAgent,ourMutualFriend,_2170111)
 I told buyAgent inBasket(shoppingAgent,ourMutualFriend,47899)
 buy(shoppingAgent,ourMutualFriend) performed successfully
 The KIF ontology has been updated

The plan is completed

The following actions have been performed:
 [buy(shoppingAgent,ourMutualFriend),putInBasket(shoppingAgent,
 bookShopGroup,ourMutualFriend),joinGroup(shoppingAgent,bookSho
 pGroup),start]

The following repairs have been performed:
 [[chooseItem(shoppingAgent,ourMutualFriend),chooseThing(shoppi
 ngAgent,ourMutualFriend),[predicateAA,[chooseThing,chooseItem]
]],[group,shoppingGroup],[precondAA,class]]]

To terminate the program, type "t."
 yes

The updated bridge rule C after finished running test case 5:

```
bridgeRule([original,update],[repairType,[repairInfo]]).
```

```
bridgeRule([interest(bookShopGroup,shoppingAgent),interest(sho  

ppingAgent,bookShopGroup)],[switchArgs,[interest,2,[shoppingGr  

oup,agent],[agent,shoppingGroup],[shoppingGroup,agent]]]).
```

```
bridgeRule([chooseThing(shoppingAgent,ourMutualFriend),chooseT  

hing(shoppingAgent,titchmarshMemoirs)],[domainA,[chooseThing,2  

,[agent,book],gardeningBook,subclass(gardeningBook,book)]]).
```

```
bridgeRule([money(shoppingAgent,dollars,100),money(shoppingAge  

nt,100)],[propositionalA,[money,3,[agent,currency,uninstantiat  

ed],currency,2]]).
```

```
bridgeRule([chooseItem(shoppingAgent,ourMutualFriend),chooseTh  

ing(shoppingAgent,ourMutualFriend)],[predicateAA,[chooseItem,2  

,[agent,book]]]).
```

C.3 The Output from Case 9

```
| ?- start.
```

```
Consulting the PLANNER...  

Consulting the ONTOLOGY UPDATER...  

Consulting ORS...
```

```

┌───────────┐
│ Goal is:   │ has(shoppingAgent,ourMutualFriend)
└───────────┘

```

Translating ...

Need to find a plan ...

This is the plan:

```
[putInBasket(shoppingAgent,aiGroup,ourMutualFriend),buy(shoppingAgent,ourMutualFriend)]
```

Executing the plan ...

```

I'm going to ask putInBasketAgent to perform
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) for me
putInBasketAgent asked me class(aiGroup,group)
I told putInBasketAgent class(aiGroup,group)
putInBasketAgent asked me
registeredMember(shoppingAgent,aiGroup)
I told putInBasketAgent
registeredMember(shoppingAgent,aiGroup)
putInBasketAgent asked me
chooseThing(shoppingAgent,ourMutualFriend)

```

Looking up a bridge rule...

Predicate found in a bridge rule : chooseThing

An unknown term has the same arity as in a bridge rule : 2

An unknown term has the same argument classes: [agent,book]

RHS term in a bridge rule :

```
chooseItem(shoppingAgent,ourMutualFriend)
```

Checking the PA's ontology...

RHS term is not found in the ontology

Checking the link between each bridge rule...

Found the related links:

```
chooseThing(shoppingAgent,ourMutualFriend) -->
chooseItem(shoppingAgent,ourMutualFriend)
chooseItem(shoppingAgent,ourMutualFriend)--
>chooseItem(shoppingAgent,ourMutualFriend,123)
```

Then it is possible to have:

```
chooseThing(shoppingAgent,ourMutualFriend) -->
chooseItem(shoppingAgent,ourMutualFriend,123)
```

Perform checking this link...

An unknown term has the same argument classes: [agent,book]

RHS term in a bridge rule :

```
chooseItem(shoppingAgent,ourMutualFriend,123)
```

Checking the PA's ontology...

Found an unknown term in a bridge rule

```
:chooseThing(shoppingAgent,ourMutualFriend)
```

I told putInBasketAgent

```
chooseThing(shoppingAgent,ourMutualFriend)
```

```
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) performed
successfully
```

I'm going to ask buyAgent to perform

```
buy(shoppingAgent,ourMutualFriend) for me
```

```
buyAgent asked me money(shoppingAgent,dollars,_582386)
```

Looking up a bridge rule...

Predicate found in a bridge rule : money

An unknown term has the same arity as in a bridge rule : 3

An unknown term has the same argument classes:

```
[agent,currency,uninstantiated]
```

RHS term in a bridge rule : money(shoppingAgent,100)

Checking the PA's ontology...

Found an unknown term in a bridge rule

```
:money(shoppingAgent,dollars,100)
```

I told buyAgent money(shoppingAgent,dollars,100)

```
buyAgent asked me
inBasket(shoppingAgent,ourMutualFriend,_586223)
I told buyAgent inBasket(shoppingAgent,ourMutualFriend,47899)
buy(shoppingAgent,ourMutualFriend) performed successfully
```

```
The KIF ontology has been updated
The following actions have been performed:
[buy(shoppingAgent,ourMutualFriend),putInBasket(shoppingAgent,
aiGroup,ourMutualFriend),start]
```

```
The following repairs have been performed:
[]
```

```
To terminate the program, type "t."
```